python 中 os 模块中文帮助文档
文章分类:Python 编程
python 中 os 模块中文帮助文档

翻 译 者 ： butalnd 翻 译 于 2010.1.7—2010.1.8 ， 个 人 博 客 ： http://butlandblog.appspot.com/
注此模块中关于 unix 中的函数大部分都被略过，翻译主要针对 WINDOWS,翻译速度很快，其中很多不足之处请多多包涵。

这个模块提供了一个轻便的方法使用要依赖操作系统的功能。 如何你只是想读或写文件，请使用 open()
,如果你想操作文件路径，请使用 os.path 模块，如果你想在命令行中，读入所有文件的所有行，请使用
fileinput 模块。使用 tempfile 模块创建临时文件和文件夹，更高级的文件和文件夹处理，请使用 shutil 模块。

os.error
内建 OSError exception 的别名。

os.name
导入依赖操作系统模块的名字。下面是目前被注册的名字:'posix', 'nt', 'mac', 'os2', 'ce', 'java', 'riscos'.

下面的 function 和 data 项是和当前的进程和用户有关

os.environ
一个 mapping 对象表示环境。例如，environ['HOME'] ，表示的你自己 home 文件夹的路径(某些平台支持，windows 不支持)
，它与 C 中的 getenv("HOME")一致。

这个 mapping 对象在 os 模块第一次导入时被创建，一般在 python 启动时，作为 site.py 处理过程的一部分。在这一次之后改变 environment 不
影响 os.environ，除非直接修改 os.environ.

注：putenv()不会直接改变 os.environ，所以最好是修改 os.environ

注:在一些平台上,包括 FreeBSD 和 Mac OS X,修改 environ 会导致内存泄露。参考 putenv()的系统文档。

如果没有提供 putenv()，mapping 的修改版本传递给合适的创建过程函数，将导致子过程使用一个修改的 environment。

如果这个平台支持 unsetenv()函数，你可以删除 mapping 中的项目。当从 os.environ 使用 pop()或 clear()删除一个项目时，unsetenv()会自动被调用（版本 2.6）。

os.chdir(path)
os.fchdir(fd)
os.getcwd()
这些函数在 Files 和 Directories 中。

os.ctermid()
返回进程控制终端的文件名。在 unix 中有效，请查看相关文档.。
os.getegid()
返回当前进程有效的 group 的 id。对应于当前进程的可执行文件的"set id "的 bit 位。在 unix 中有效，请查看相关文档.。
os.geteuid()
返回当前进程有效的 user 的 id。在 unix 中有效，请查看相关文档.。
os.getgid()
返回当前进程当前 group 的 id。在 unix 中有效，请查看相关文档.。
os.getgroups()
返回当前进程支持的 groups 的 id 列表。在 unix 中有效，请查看相关文档.。
os.getlogin()
返回进程控制终端登陆用户的名字。在大多情况下它比使用 environment 变量 LOGNAME 来得到用户名，或使用 pwd.getpwuid(os.getuid())[0] 得到当前有效用户 id 的登陆名更为有效。在 unix 中有效，请查看相关文档.。

os.getpgid(pid)
返回 pid 进程的 group id.如果 pid 为 0,返回当前进程的 group id。在 unix 中有效，请查看相关文档.。

os.getpgrp()
返回当前进程组的 id.在 unix 中有效，请查看相关文档.。

os.getpid()
返回当前进程的 id.在 unix，Windows 中有效。

os.getppid()
返回当前父进程的 id.在 unix 中有效，请查看相关文档.。
os.getuid()
返回当前当前进程用户的 id.在 unix 中有效，请查看相关文档.。
os.getenv(varname[, value])
返回 environment 变量 varname 的值，如果 value 不存在，默认为 None.在大多版本的 unix，Windows 中有效。

os.putenv(varname, value)

设置 varname 环境变量为 value 值。此改变影响以 os.system(), popen() 或 fork()

和 execv()启动的子进程。在大多版本的 unix，Windows 中有效。

当支持 putenv()时，在 os.environ 分配项目时，自动调用合适的 putenv()。然而，调用 putenv() 不会更新 os.environ，所以直接设置 os.environ 的项。

os.setegid(egid)
设置当前进程有效组的 id.在 unix 中有效，请查看相关文档.。

os.seteuid(euid)
设置当前进程有效用户的 id.在 unix 中有效，请查看相关文档.。
os.setgid(gid)
设置当前进程组的 id.在 unix 中有效，请查看相关文档.。
os.setgroups(groups)
设置当前进程支持的 groups id 列表。groups 必须是个列表，每个元素必须是个整数，这个操作只对超级用户有效，在 unix 中有效，请查看相关文档.。

os.setpgrp()
调用 system 的 setpgrp()或 setpgrp(0, 0)() ，依赖于使用的是哪个版本的 system. 请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。
os.setpgid(pid, pgrp)
调用 system 的 setpgid()设置 pid 进程 group 的 id 为 pgrp.请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。
os.setreuid(ruid, euid)
设置当前 process 当前 和有效的用户 id. 在 unix 中有效，请查看相关文档.。
os.setregid(rgid, egid)
设置当前 process 当前 和有效的组 id. 在 unix 中有效，请查看相关文档.。
os.getsid(pid)
调用 system 的 getsid(). 请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。

os.setsid()
调用 system 的 setsid().请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。
os.setuid(uid)
设置当前 user id. 在 unix 中有效，请查看相关文档.。

os.strerror(code)
返回程序中错误 code 的错误信息。在某些平台上，当给一个未知的 code,strerror()返回 NULL,将抛出 ValueError。在 unix，Windows 中有效。
os.umask(mask)
设置当前权限掩码，同时返回先前的权限掩码。在 unix，Windows 中有效。

os.fdopen(fd[, mode[, bufsize]])
返回一个文件描述符号为 fd 的打开的文件对象。mode 和 bufsize 参数，和内建的 open()函数是同一个意思。在 unix，Windows 中有效。
mode 必须以'r', 'w',或者'a'开头，否则抛出 ValueError.

以'a'开头的 mode，文件描述符中 O_APPEND 位已设置.

os.popen(command[, mode[, bufsize]])
给或从一个 command 打开一个管理。返回一个打开的连接到管道文件对象，文件对象可以读或写，在于模式是'r'(默认) 或'w'，bufsize 参数，和内建的 open()函数是同一个意思。 command 返回的状态 (在 wait()函数中编码) 和调用文件对象的 close()返回值一样，除非返回值是 0(无错误终止)，返回 None . 在 unix，Windows 中有效。

在 2.6 版本中已抛弃. 使用 subprocess 模块.

os.tmpfile()
返回一个打开的模式为(w+b)的文件对象 .这文件对象没有文件夹入口，没有文件描述符，将会自动删除. 在 unix，Windows 中有效。

从 version 2.6 起: 所有的 popen*()函数已抛弃. 使用 subprocess 模块.

os.popen2(cmd[, mode[, bufsize]])

os.popen3(cmd[, mode[, bufsize]])

os.popen4(cmd[, mode[, bufsize]])

16.1.3. 文件描述符操作
这些函数操作使用文件描述符引用的 I/O stream。
文件描述符是与当前进程打开的文件相对应的一些小整数. 例如标准输入的通常文件描述符中 0，标准输出是 1,标准错误是 2. 进程打开的更多文件将被分配为 3，4，5,等. 这“文件描述符”有一点迷惑性;在 Unix 平台上，socket 和 pipe 通常也使用文件描述符引用.

os.close(fd)
关闭文件描述符 fd. 在 unix，Windows 中有效。

这函数是为低层的 I/O 服务的，应用在 os.open()或 pipe()返回的文件描述符上。关闭一个由内建函数 open()或 popen()或 fdopen()打开的文件对象，使用 close()方法。
os.closerange(fd_low, fd_high)
关闭从 fd_low（包含）到 fd_high（不包含）所有的文件描述符，忽略错误。在 unix，Windows 中有效。
等同于：

```
for fd in xrange(fd_low, fd_high):
    try:
```

```
        os.close(fd)
    except OSError:
        pass
```

os.dup(fd)
返回文件描述符 fd 的 cope．在 unix，Windows 中有效。
os.dup2(fd, fd2)
复制文件描述符 fd 到 fd2，如果有需要首先关闭 fd2．在 unix，Windows 中有效。
os.fchmod(fd, mode)
改变文件描述符为 fd 的文件’mode‘为 mode．查看 chmod()文档 中 mode 的值．在 unix 中有效，请查看相关文档.。

version 2.6 中新增.

os.fchown(fd, uid, gid)
改变文件描述符为 fd 的文件的所有者和 group 的 id 为 uid 和 gid．如果不想它们中的一个，就设置为-1．在 unix 中有效，请查看相关文档.。

version 2.6 中新增.

os.fdatasync(fd)
强制将文件描述符为 fd 的文件写入硬盘．不强制更新 metadata．在 unix 中有效,请查看相关文档.。

注：在 MacOS 中无效.

os.fpathconf(fd, name)
返回一个打开的文件的系统配置信息。name 为检索的系统配置的值，它也许是一个定义系统值的字符串，这些名字在很多标准中指定（POSIX.1，Unix 95，Unix 98，和其它）。一些平台也定义了一些额外的名字。这些名字在主操作系统上 pathconf_names 的字典中。对于不在 pathconf_names 中的配置变量，传递一个数字作为名字，也是可以接受的。 在 unix 中有效，请查看相关文档.。

如果 name 是一个字符串或者未知的，将抛出 ValueError。如果 name 是一个特别的值，在系统上不支持，即使它包含在 pathconf_names 中，将会抛出错误数字为 errno.EINVAL 的 OSError。

os.fstat(fd)
返回文件描述符 fd 的状态，像 stat()。在 unix，Windows 中有效。
os.fstatvfs(fd)
返回包含文件描述符 fd 的文件的文件系统的信息，像 statvfs()．在 unix 中有效，请查看相关文档.。
os.fsync(fd)

强制将文件描述符为 fd 的文件写入硬盘.在 Unix，将调用 fsync()函数;在 Windows，调用 _commit()函数.

如果你准备操作一个 Python 文件对象 f，首先 f.flush(),然后 os.fsync(f.fileno()),确保与 f 相关的所有内存都写入了硬盘.在 unix，Windows 中有效。

os.ftruncate(fd, length)
裁剪文件描述符 fd 对应的文件，所以它最大不能超过文件大小．在 unix 中有效，请查看相关文档.。
os.isatty(fd)
如果文件描述符 fd 是打开的,同时与 tty(-like)设备相连,则返回 true，否则 False．在 unix 中有效，请查看相关文档.。
os.lseek(fd, pos, how)
设置文件描述符 fd 当前位置为 pos，how 方式修改：SEEK_SET 或者 0 设置从文件开始的计算的 pos；SEEK_CUR 或者 1 则从当前位置计算；os.SEEK_END 或者 2 则从文件尾部开始．在 unix，Windows 中有效。
os.open(file, flags[, mode])
打开 file 同时根据 flags 设置变量 flags ，如果有 mode,则设置它的 mode．默认的 mode 是 0777 (八进制)，当前掩码值是 first masked out．返回刚打开的文件描述符．在 unix，Windows 中有效。

flag 和 mode 值，请查看 C 运行时文档；flag 常数(像 O_RDONLY and O_WRONLY)在这个模块中也定义了（在下面）.

注:这函数是打算为低层 I/O 服务的.正常的使用，使用内建函数 open(),返回 read()和 write() 等方法创建的文件对象.包装文件描述符为“文件对象”，使用 fdopen().

os.openpty()
在一些 Unix 平台上有效，请查看相关文档.

os.pipe()
创建一个管道．返回一对文件描述符(r, w) 分别为读和写．在 unix，Windows 中有效。
os.read(fd, n)
从文件描述符 fd 中读取最多 n 个字节．返回包含读取字节的 string．文件描述符 fd 对应文件已达到结尾，返回一个空 string．在 unix，Windows 中有效。

注:这函数是打算为低层 I/O 服务的 ，同时必须应用在 os.open()或者 pipe()函数返回的文件描述符．读取内建函数 open()或者 by popen()或者 fdopen(),或者 sys.stdin 返回的一个“文件对象” ，使用它的 read()或者 readline()方法.

os.tcgetpgrp(fd)
在 unix 中有效，请查看相关文档.。
os.tcsetpgrp(fd, pg)
在 unix 中有效，请查看相关文档.。

os.ttyname(fd)
在 unix 中有效，请查看相关文档.。
os.write(fd, str)
写入字符串到文件描述符 fd 中. 返回实际写入的字符串长度. 在 unix,Windows 中有效。

注:这函数是打算为低层 I/O 服务的 ，同时必须应用在 os.open()或者 pipe()函数返回的文件描述符. 读取内建函数 open()或者 by popen()或者 fdopen(),或者 sys.stdin 返回的一个“文件对象” , 使用它的 read()或者 readline()方法.

下面的常数是 open()函数的 flags 参数选项. 它们可以使用 bitwise 合并或者 operator |. 它们中的一些常数并不是在所有平台都有效. 它们更多使用请查看相关资料，在 unix 上参考 open(2) 手册页面，windows 上 http://msdn.microsoft.com/en-us/library/z0kc8e3z.aspx.

os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC
这些常数在 Unix and Windows 上有效.
os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_SHLOCK
os.O_EXLOCK
这些常数仅在 Unix 上有效.
os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM
os.O_SEQUENTIAL
os.O_TEXT
这些常数仅在 Windows 上有效.
os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME

这些常数是 GNU 扩展，如果没有在 C 库声明刚没有.
os.SEEK_SET
os.SEEK_CUR
os.SEEK_END
lseek()函数的参数. 它们的值分别是 0, 1,和 2. 在 Unix and Windows 上有效.

版本 2.5 新增.

16.1.4. 文件和文件夹
os.access(path, mode)
使用现在的uid/gid尝试访问path. 注大部分操作使用有效的uid/gid, 因此运行环境可以在 suid/sgid 环境尝试，如果用户有权访问 path. mode 为 F_OK，测试存在的 path, 或者它可以是包含 R_OK, W_OK 和 X_OK 或者 R_OK, W_OK 和 X_OK 其中之一或者更多.如果允许访问返回 True , 否则返回 False. 查看 Unix 手册 access(2) 获取更多信息. 在 unix，Windows 中有效。

注:使用access()去测试用户是否授权. 在实际使用open()打开一个文件前测试会创建一个安全漏洞前 , 因为用户会利用这短暂时间在检测和打开这个文件去修改它.

注:即使 access()表明它将 succeed，但 I/O 操作也可能会失败, 如网络文件系统.

os.F_OK
作为 access()的 mode 参数，测试 path 是否存在.
os.R_OK
包含在 access()的 mode 参数中 ， 测试 path 是否可读.
os.W_OK
包含在 access()的 mode 参数中 ，测试 path 是否可写.
os.X_OK
包含在 access()的 mode 参数中 ，测试 path 是否可执行..
os.chdir(path)
改变当前工作目录. 在 unix，Windows 中有效。

os.fchdir(fd)
在 unix 中有效，请查看相关文档.。
os.getcwd()
返回当前工作目录的字符串， 在 unix，Windows 中有效。
os.getcwdu()
返回一个当前工作目录的 Unicode 对象. 在 unix，Windows 中有效。
os.chflags(path, flags)
在 unix 中有效，请查看相关文档.。

os.chroot(path)
在 unix 中有效，请查看相关文档.。

os.chmod(path, mode)
改变path的mode到数字mode. mode为下面中的一个 (在stat模块中定义)或者bitwise或者它们的组合:

?stat.S_ISUID
?stat.S_ISGID
?stat.S_ENFMT
?stat.S_ISVTX
?stat.S_IREAD
?stat.S_IWRITE
?stat.S_IEXEC
?stat.S_IRWXU
?stat.S_IRUSR
?stat.S_IWUSR
?stat.S_IXUSR
?stat.S_IRWXG
?stat.S_IRGRP
?stat.S_IWGRP
?stat.S_IXGRP
?stat.S_IRWXO
?stat.S_IROTH
?stat.S_IWOTH
?stat.S_IXOTH
在unix，Windows中有效。

注:尽管Windows支持chmod(), 你只可以使用它设置只读 flag (通过stat.S_IWRITE和stat.S_IREAD 常数或者一个相对应的整数)。所有其它的 bits都忽略了.

os.chown(path, uid, gid)
在unix中有效，请查看相关文档.。
os.lchflags(path, flags)
在unix中有效，请查看相关文档.。

新增version 2.6.

os.lchmod(path, mode)
在unix中有效，请查看相关文档.。

新增version 2.6.

os.lchown(path, uid, gid)
在unix中有效，请查看相关文档.。

新增 version 2.3.

os.link(source, link_name)
在 unix 中有效，请查看相关文档.。
os.listdir(path)
返回 path 指定的文件夹包含的文件或文件夹的名字的列表．这个列表以字母顺序．它不包括 '.' 和'..' 即使它在文件夹中．在 unix，Windows 中有效。

Changed in version 2.3:在 Windows NT/2k/XP 和 Unix，如果文件夹是一个 Unicode object，结果将是 Unicode objects 列表．不能解码的文件名将仍然作为 string objects 返回.

os.lstat(path)
像 stat(),但是没有符号链接．这是 stat()的别名 在某些平台上,例如 Windows.
os.mkfifo(path[, mode])
在 unix 中有效，请查看相关文档.。

os.mknod(filename[, mode=0600, device])
创建一个名为 filename 文件系统节点（文件，设备特别文件或者命名 pipe）。 mode 指定创建或使用节点的权限，组合 (或者 bitwise) stat.S_IFREG, stat.S_IFCHR, stat.S_IFBLK, 和 stat.S_IFIFO (这些常数在 stat 模块). 对于 stat.S_IFCHR 和 stat.S_IFBLK, 设备定义了 最新创建的设备特殊文件 (可能使用 os.makedev()),其它都将忽略.

新增 version 2.3.

os.major(device)
从原始的设备号中提取设备 major 号码 (使用 stat 中的 st_dev 或者 st_rdev field).

新增 version 2.3.

os.minor(device)
从原始的设备号中提取设备 minor 号码 (使用 stat 中的 st_dev 或者 st_rdev field).

新增 version 2.3.

os.makedev(major, minor)
以 major 和 minor 设备号组成一个原始设备号.

新增 version 2.3.

os.mkdir(path[, mode])

以数字 mode 的 mode 创建一个名为 path 的文件夹.默认的 mode 是 0777 (八进制). 在有些平台上, mode 是忽略的. 当使用时。这当前的掩码值是 first masked out. 在 unix, Windows 中有效。

也可以创建临时文件夹; 查看 tempfile 模块 tempfile.mkdtemp()函数.

os.makedirs(path[, mode])
递归文件夹创建函数。像 mkdir(), 但创建的所有 intermediate-level 文件夹需要包含子文件夹. 抛出一个 error exception 如果子文件夹已经存在或者不能创建. 默认的 mode 是 0777 (八进制). 在有些平台上, mode 是忽略的.当使用时。这当前的掩码值是 first masked out。

注:
makedirs()变得迷惑 如果路径元素包含 os.pardir.
现在可以正确处理 UNC 路径.

os.pathconf(path, name)
在 unix 中有效，请查看相关文档.。

os.pathconf_names
在 unix 中有效，请查看相关文档.。
os.readlink(path)
在 unix 中有效，请查看相关文档.。

os.remove(path)
删除路径为 path 的文件.如果 path 是一个文件夹,将抛出 OSError; 查看下面的 rmdir()删除一个 directory. 这和下面的 unlink()函数文档是一样的.在 Windows,尝试删除一个正在使用的文件将抛出一个 exception;在 Unix,directory 入口会被删除，但分配给文件的存储是无效的，直到原来的文件不再使用. 在 unix，Windows 中有效。
os.removedirs(path)
递归删除 directorie. 像 rmdir(), 如果子文件夹成功删除, removedirs()才尝试它们的父文件夹,直到抛出一个 error(它基本上被忽略,因为它一般意味着你文件夹不为空). 例如, os.removedirs('foo/bar/baz') 将首先删除 'foo/bar/baz', 然后删除 'foo/bar' 和 'foo' 如果它们是空的. 如果子文件夹不能被成功删除,将抛出 OSError .

新增 version 1.5.2.

os.rename(src, dst)
重命名 file 或者 directory src 到 dst.如果 dst 是一个存在的 directory, 将抛出 OSError. 在 Unix, 如果 dst 在存且是一个 file,如果用户有权限的话，它将被安静的替换. 操作将会失败在某些 Unix 中如果 src 和 dst 在不同的文件系统中. 如果成功, 这命名操作将会是一个原子操作 (这是 POSIX 需要). 在 Windows 上, 如果 dst 已经存在, 将抛出 OSError，即使它是一个文件. 在 unix，Windows 中有效。
os.renames(old, new)

递归重命名文件夹或者文件。像 rename()

新增 version 1.5.2.

os.rmdir(path)
删除 path 文件夹．仅当这文件夹是空的才可以，否则，抛出 OSError．要删除整个文件夹树，可以使用 shutil.rmtree()．在 unix，Windows 中有效。
os.stat(path)
执行一个 stat()系统调用在给定的 path 上．返回值是一个对象，属性与 stat 结构成员有关：st_mode（保护位），st_ino（inode number），st_dev（device），st_nlink（number of hard links），st_uid（所有用户的 id），st_gid（所有者 group id），st_size（文件大小，以位为单位），st_atime（最近访问的时间），st_mtime（最近修改的时间），st_ctime（依赖于平台;在 Unix 上是 metadata 最近改变的时间,或者在 Windows 上是创建时间）:

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511L, 769L, 1, 1032, 100, 926L, 1105022698,1105022732,
1105022732)
>>> statinfo.st_size
926L
>>>
```

如果 stat_float_times()返回 True,time 值是 floats,以 second 进行计算．一秒的小数部分也会显示出来，如果系统支持．在 Mac OS，时间常常是 floats.查看 stat_float_times() 获取更多信息.

在一些 Unix 系统上(例如 Linux)，下面的属性也许是有效的：st_blocks（为文件分配了多少块），st_blksize（文件系统 blocksize），st_rdev（设备型号如果是一个 inode 设备）．st_flags（用户为文件定义的 flags）.

在 unix，Windows 中有效。

os.stat_float_times([newvalue])
决定 stat_result 是否以 float 对象显示时间戳.

os.statvfs(path)
在 unix 中有效，请查看相关文档.。

os.symlink(source, link_name)

在 unix 中有效，请查看相关文档.。
os.tempnam([dir[, prefix]])
为创建一个临时文件返回一个唯一的 path。在 Windows 使用 TMP . 依赖于使用的 C 库;
警告:
使用 tempnam() 对于 symlink 攻击是一个漏洞; 考虑使用 tmpfile()代替.

在 unix，Windows 中有效。

os.tmpnam()
为创建一个临时文件返回一个唯一的 path.
Warning:
使用 tempnam() 对于 symlink 攻击是一个漏洞; 考虑使用 tmpfile()代替.

在 unix，Windows 中有效。

os.TMP_MAX
tmpnam() 将产生唯一名字的最大数值.
os.unlink(path)
删除 file 路径. 与 remove()相同; 在 unix，Windows 中有效。
os.utime(path, times)
返回指定的 path 文件的访问和修改的时间。如果时间是 None, 则文件的访问和修改设为当前时间 。 否则, 时间是一个 2-tuple 数字, (atime, mtime) 用来分别作为访问和修改的时间.
在 unix，Windows 中有效。

os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])
输出在文件夹中的文件名通过在树中游走，向上或者向下.在根目录下的每一个文件夹(包含它自己), 产生 3-tuple (dirpath, dirnames, filenames)【文件夹路径, 文件夹名字, 文件名】.

dirpath 是一个字符串, directory 的路径. dirnames 在 dirpath 中子文件夹的列表 (不包括 '.' '..'). filenames 文件是在 dirpath 不包含子文件夹的文件名的列表. 注：列表中的 names 不包含 path.为获得 dirpath 中的一个文件或者文件夹的完整路径 (以项目录开始)或者, 操作 os.path.join(dirpath, name).

如果 optional 参数 topdown 为 True 或者 not 指定, 一个 directory 的 3-tuple 将比它的任何子文件夹的 3-tuple 先产生 (directories 自上而下).如果 topdown 为 False, 一个 directory 的 3-tuple 将比它的任何子文件夹的 3-tuple 后产生 (directories 自下而上)。

当 topdown 为 True,调用者可以修改列表中列出的文件夹名字(也可以使用 del 或者 slice), walk() 仅仅递归每一个包含在 dirnames 中的子文件夹; 可以减少查询, 利用访问的特殊顺序,或者甚至 告诉 walk() 关于文件夹的创建者或者重命名在它重新 walk() 之前.修改文件名当 topdown 为 False 时是无效的, 因为在 bottom-up 模式中在

dirnames 中的 directories 比 dirpath 它自己先产生 .

默认 listdir()的 errors 将被忽略. 如果 optional 参数 onerror 被指定,它应该是一个函数; 它调用时有一个参数, 一个 OSError 实例. 报告这错误后, 继续 walk,或者抛出 exception 终止 walk.注意 filename 是可用的的 , exception 对象的 filename 属性.

默认, walk() 不会进入符号链接 .

新增 version 2.6:

注:如果你传入一个相对的 pathname, 不要在 walk()执行过程中改变当前文件夹. walk()不会改变改前文件夹, 同时确保它的调用者也不会改变.

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print root, "consumes",
    print sum(getsize(join(root, name)) for name in files),
    print "bytes in", len(files), "non-directory files"
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories


# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
        os.rmdir(os.path.join(root, name))
```

新增 version 2.3.

使用函数来创建和管理进程。
使用 exec*() 函数使用 arguments 列表来载入新程序. 在每个例子, 一个用户敲入一个命令行中的第一个参数传递给程序作为它自己的名字而不是作为参数. 对于 C 程序员来说, 它是传递给 main()的 argv[0]. 例如, os.execv('/bin/echo', ['foo', 'bar']) 将

仅仅在标准输出上打印 bar; foo 将被忽略.

os.abort()
产生一个 SIGABRT 标识到当前的进程.在 Unix,这默认的行为是产生一个主要的 dump;在 Windows,这进程立即返回退出以一个状态码为3. 程序使用 signal.signal()来注册一个 SIGABRT 将导致不同的行为. 在 unix，Windows 中有效。
os.execl(path, arg0, arg1, ...)
os.execle(path, arg0, arg1, ..., env)
os.execlp(file, arg0, arg1, ...)
os.execlpe(file, arg0, arg1, ..., env)
os.execv(path, args)
os.execve(path, args, env)
os.execvp(file, args)
os.execvpe(file, args, env)
这些函数将执行一个新程序，替换当前进程；他们没有返回.在Unix,新的执行体载入到当前的进程，同时将和当前的调用者有相同的 id. 将报告 Errors 当抛出 OSError 时.

当前的进程立即被替代. 打开文件对象和描述符不会被刷新，如果在这些打开的文件中有数据缓冲区， 应该在调用 exec*() 函数之前，使用 sys.stdout.flush()或者 os.fsync()flush 它们 .

在 unix，Windows 中有效。

os._exit(n)
使用状态 n 退出系统,没有调用清理函数，刷新缓冲区。 在 unix，Windows 中有效。

注:标准退出的方法是 sys.exit(n). _exit()一般使用于 fork()产生的子进程中.

os.EX_OK
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_USAGE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_DATAERR
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOINPUT

在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOUSER
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOHOST
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_UNAVAILABLE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_SOFTWARE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_OSERR
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_OSFILE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_CANTCREAT
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_IOERR
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_TEMPFAIL
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_PROTOCOL
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOPERM
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_CONFIG
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOTFOUND
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.fork()
在 unix 中有效，请查看相关文档.。

os.forkpty()
在一些 unix 中有效，请查看相关文档
os.kill(pid, sig)
在 unix 中有效，请查看相关文档.。

os.killpg(pgid, sig)
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.nice(increment)
在 unix 中有效，请查看相关文档.。
os.plock(op)
在 unix 中有效，请查看相关文档.。
os.popen(...)
os.popen2(...)

os.popen3(...)
os.popen4(...)
运行子进程，返回交流的打开的管道.这些函数在前面创建文件对象时介绍过.
os.spawnl(mode, path, ...)
os.spawnle(mode, path, ..., env)
os.spawnlp(mode, file, ...)
os.spawnlpe(mode, file, ..., env)
os.spawnv(mode, path, args)
os.spawnve(mode, path, args, env)
os.spawnvp(mode, file, args)
os.spawnvpe(mode, file, args, env)
在新进程中执行程序 path
(请使用 subprocess 模块)

如果模式是 P_NOWAIT，返回新进程的 id;如果模式是 P_WAIT,返回进程退出时的状态码。如果正常退出,或者-signal，当 signal 是 killed.在 Windows，进程 id 实际上是 process 的 handle,所它可以使用于 waitpid()函数.

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

在 unix，Windows 中有效。

新增 version 1.6.

os.P_NOWAIT
os.P_NOWAITO
spawn*()族函数参数 mode 的可选值．如果给出其中任一个值，新进程一创建完成，spawn*()函数将立即返回 ,返回进程 id 的值．在 unix，Windows 中有效。

新增 version 1.6.

os.P_WAIT
spawn*()族函数参数 mode 的可能值．如果将它赋值给 mode，spawn*() 函数不返回，直接运行结束 以及如果运行成功 ,将返回进程的退出码,或者如果 signal 杀掉了这个进程，将返回-signal．在 unix，Windows 中有效。

新增 version 1.6.

os.P_DETACH
os.P_OVERLAY

spawn*()族函数参数 mode 的可选值。P_DETACH 和 P_NOWAIT 很相似，但是新进程依附在调用进程的 console 上。如果使用了 P_OVERLAY，当前进程将被替换，spawn*()函数将无返回 。在 Windows 上有效。
新增 version 1.6.

os.startfile(path[, operation])
以相关的程序打开文件.

当 operation 没有指定或者'open'，这操作就像在 Windows Explorer 双击文件,或者将这个文件作为交互命令行中 start 命令的参数:与文件扩展相关的程序打开文件。

当指定另外操作时，它必须是“command verb”它指定应该对文件做什么.像 Microsoft 的'print' 'edit' (作用在文件上) 'explore' and 'find' (作用在文件夹上).

startfile()只要相关的应该程序一启动就返回。 没有选项等待应用程序关闭，没有方法接收应用程序退出的状态. path 参数与当前 directory 相关. 如果你想使用一个绝对路径，确保第一个字符不是 ('/')。 使用 os.path.normpath() 函数确保路径对于 win32 来说是正确的. 在 Windows 上有效.

新增 version 2.0.

新增 version 2.5: The operation parameter.

os.system(command)
在 shell 中执行 string 命令. 这是使用标准 C 函数 system()，有同样的限制. 例如改变 sys.stdin，不影响命令执行环境.

在 Unix，请查看相当文档.

在 Windows，返回值是在 shell 运行命令的返回值。

在 unix，Windows 中有效。

subprocess 模块提供了一个更强大的功能产生新进程和接收它们的结果;

os.times()
返回一个 5-tuple 的浮点数字， 表示(处理器或者其它)累积时间，以秒为单位. items 为:用户时间，系统 time，子用户 time，子系统 time，和从过去一个固定的点真实流逝的时间. 在 unix，Windows 中有效。在 Windows，仅仅填充开始两项，其它都为 0.
os.wait()
在 unix 中有效，请查看相关文档.。
os.waitpid(pid, options)
Unix:等待一个指定的 pid 的子进程完成，返回一个 tuple 返回它的进程 id 和退出状态 . 一般情况下 option 设为 0.

更强细请查看相关文档

在 Windows: 等待一个指定的 pid 的进程完成, 返回一个 tuple 返回它的进程 id 和退出状态向左移动了 8 位 。 如果 pid 小于或等于 0 没有特别的意思,将抛出 exception. integer options 没有任何影响. pid 可以指向任何进程的 id,不一定是子进程的 id.

os.wait3([options])
在 unix 中有效，请查看相关文档.。

新增 version 2.5.

os.wait4(pid, options)
在 unix 中有效，请查看相关文档.。

新增 version 2.5.

os.WNOHANG
在 unix 中有效，请查看相关文档.。
os.WCONTINUED
在某些 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.WUNTRACED
在 unix 中有效，请查看相关文档.。

os.WCOREDUMP(status)
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.WIFCONTINUED(status)
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.WIFSTOPPED(status)
在 unix 中有效，请查看相关文档.。
os.WIFSIGNALED(status)
在 unix 中有效，请查看相关文档.。
os.WIFEXITED(status)
在 unix 中有效，请查看相关文档.。

os.WEXITSTATUS(status)
在 unix 中有效，请查看相关文档.。
os.WSTOPSIG(status)
在 unix 中有效，请查看相关文档.。
os.WTERMSIG(status)
在 unix 中有效，请查看相关文档.。
16.1.6. Miscellaneous System Information
os.confstr(name)
在 unix 中有效，请查看相关文档.。

os.confstr_names
在 unix 中有效，请查看相关文档.。
os.getloadavg()
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.sysconf(name)
在 unix 中有效，请查看相关文档.。

os.curdir
操作系统用此常数字符串作为当前文件夹的引用。
os.pardir
操作系统用此常数字符串作为父文件夹的引用。
os.sep
系统使用此字符来分割路径。
os.altsep
系统使用另外一个字符来分割路径，如果只有一个分割字符存在，则是 None.
os.extsep
分割基本文件名和扩展名的字符。

新增 version 2.2.

os.pathsep
系统使用此字符来分割搜索路径（像 PATH），例如 POSIX 上':',Windows 上的';'，也可以通过 os.path
os.defpath
默认的搜索路径用作 exec*p*()和 spawn*p*()如果环境没有 'PATH'. 也可以通过 os.path.
os.linesep
当前平台上的换行符字符串. 在 POSIX 上是'\n',或者 在 Windows 上是'\r\n' . 不要使用 os.linesep 作为换行符，当写入文本文件时 (默认); 使用'\n'代替, 在所有平台上.

os.devnull
空设备的文件路径.例如:POSIX 上 '/dev/null' . 也可以通过 os.path.

新增 version 2.4.

16.1.7. 其它 函数
os.urandom(n)
返回 n 个随机 byte 值的 string，作为加密使用
python 中 os 模块中文帮助文档
文章分类:Python 编程
python 中 os 模块中文帮助文档

翻 译 者 ： butalnd 翻 译 于 2010.1.7—2010.1.8 ， 个 人 博 客 ： http://butlandblog.appspot.com/
注此模块中关于 unix 中的函数大部分都被略过，翻译主要针对 WINDOWS,翻译速度很快，其中很多不足之处请多多包涵。

这个模块提供了一个轻便的方法使用要依赖操作系统的功能。 如何你只是想读或写文件，请使用 open()
,如果你想操作文件路径，请使用 os.path 模块，如果你想在命令行中，读入所有文件的所有行，请使用
fileinput 模块。使用 tempfile 模块创建临时文件和文件夹，更高级的文件和文件夹处理，请使用 shutil 模块。

os.error
内建 OSError exception 的别名。

os.name
导入依赖操作系统模块的名字。下面是目前被注册的名字:'posix', 'nt', 'mac', 'os2', 'ce', 'java', 'riscos'.

下面的 function 和 data 项是和当前的进程和用户有关

os.environ
一个 mapping 对象表示环境。例如，environ['HOME'] ，表示的你自己 home 文件夹的路径(某些平台支持，windows 不支持)
，它与 C 中的 getenv("HOME")一致。

这个 mapping 对象在 os 模块第一次导入时被创建，一般在 python 启动时，作为 site.py 处理过程的一部分。在这一次之后改变 environment 不
影响 os.environ，除非直接修改 os.environ.

注：putenv()不会直接改变 os.environ，所以最好是修改 os.environ

注:在一些平台上,包括 FreeBSD 和 Mac OS X,修改 environ 会导致内存泄露。参考 putenv()的系统文档。

如果没有提供 putenv()，mapping 的修改版本传递给合适的创建过程函数，将导致子过程使用一个修改的 environment。

如果这个平台支持 unsetenv()函数，你可以删除 mapping 中的项目。当从 os.environ 使用 pop()或 clear()删除一个项目时，unsetenv()会自动被调用（版本 2.6）。

os.chdir(path)
os.fchdir(fd)
os.getcwd()
这些函数在 Files 和 Directories 中。

os.ctermid()
返回进程控制终端的文件名。在 unix 中有效，请查看相关文档.。
os.getegid()
返回当前进程有效的 group 的 id。对应于当前进程的可执行文件的"set id "的 bit 位。在 unix 中有效，请查看相关文档.。
os.geteuid()
返回当前进程有效的 user 的 id。在 unix 中有效，请查看相关文档.。
os.getgid()
返回当前进程当前 group 的 id。在 unix 中有效，请查看相关文档.。
os.getgroups()
返回当前进程支持的 groups 的 id 列表。在 unix 中有效，请查看相关文档.。
os.getlogin()
返回进程控制终端登陆用户的名字。在大多情况下它比使用 environment 变量 LOGNAME 来得到用户名，或使用 pwd.getpwuid(os.getuid())[0] 得到当前有效用户 id 的登陆名更为有效。在 unix 中有效，请查看相关文档.。

os.getpgid(pid)
返回 pid 进程的 group id.如果 pid 为 0,返回当前进程的 group id。在 unix 中有效，请查看相关文档.。

os.getpgrp()
返回当前进程组的 id.在 unix 中有效，请查看相关文档.。

os.getpid()
返回当前进程的 id.在 unix，Windows 中有效。

os.getppid()
返回当前父进程的 id.在 unix 中有效，请查看相关文档.。
os.getuid()

返回当前当前进程用户的 id.在 unix 中有效，请查看相关文档.。
os.getenv(varname[, value])
返回 environment 变量 varname 的值，如果 value 不存在，默认为 None.在大多版本的 unix，Windows 中有效。

os.putenv(varname, value)

设置 varname 环境变量为 value 值。此改变影响以 os.system(), popen() 或 fork() 和 execv()启动的子进程。在大多版本的 unix，Windows 中有效。

当支持 putenv()时，在 os.environ 分配项目时，自动调用合适的 putenv()。然而，调用 putenv() 不会更新 os.environ，所以直接设置 os.environ 的项。

os.setegid(egid)
设置当前进程有效组的 id.在 unix 中有效，请查看相关文档.。

os.seteuid(euid)
设置当前进程有效用户的 id.在 unix 中有效，请查看相关文档.。
os.setgid(gid)
设置当前进程组的 id.在 unix 中有效，请查看相关文档.。
os.setgroups(groups)
设置当前进程支持的 groups id 列表。groups 必须是个列表，每个元素必须是个整数，这个操作只对超级用户有效，在 unix 中有效，请查看相关文档.。

os.setpgrp()
调用 system 的 setpgrp()或 setpgrp(0, 0)() ，依赖于使用的是哪个版本的 system. 请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。
os.setpgid(pid, pgrp)
调用 system 的 setpgid()设置 pid 进程 group 的 id 为 pgrp.请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。
os.setreuid(ruid, euid)
设置当前 process 当前 和有效的用户 id. 在 unix 中有效，请查看相关文档.。
os.setregid(rgid, egid)
设置当前 process 当前 和有效的组 id. 在 unix 中有效，请查看相关文档.。
os.getsid(pid)
调用 system 的 getsid(). 请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。

os.setsid()
调用 system 的 setsid().请查看 Unix 手册. 在 unix 中有效，请查看相关文档.。
os.setuid(uid)
设置当前 user id. 在 unix 中有效，请查看相关文档.。

os.strerror(code)
返回程序中错误 code 的错误信息。在某些平台上，当给一个未知的 code,strerror()返

回 NULL,将抛出 ValueError。在 unix，Windows 中有效。
os.umask(mask)
设置当前权限掩码，同时返回先前的权限掩码。在 unix，Windows 中有效。

os.fdopen(fd[, mode[, bufsize]])
返回一个文件描述符号为 fd 的打开的文件对象。mode 和 bufsize 参数，和内建的 open()函数是同一个意思。在 unix，Windows 中有效。
mode 必须以'r', 'w',或者'a'开头，否则抛出 ValueError.

以'a'开头的 mode，文件描述符中 O_APPEND 位已设置.

os.popen(command[, mode[, bufsize]])
给或从一个 command 打开一个管理。返回一个打开的连接到管道文件对象，文件对象可以读或写，在于模式是'r'(默认) 或'w'，bufsize 参数，和内建的 open()函数是同一个意思。 command 返回的状态 (在 wait()函数中编码) 和调用文件对象的 close()返回值一样，除非返回值是 0(无错误终止)，返回 None . 在 unix，Windows 中有效。

在 2.6 版本中已抛弃. 使用 subprocess 模块.

os.tmpfile()
返回一个打开的模式为(w+b)的文件对象 .这文件对象没有文件夹入口，没有文件描述符，将会自动删除. 在 unix，Windows 中有效。

从 version 2.6 起: 所有的 popen*()函数已抛弃. 使用 subprocess 模块.

os.popen2(cmd[, mode[, bufsize]])

os.popen3(cmd[, mode[, bufsize]])

os.popen4(cmd[, mode[, bufsize]])

16.1.3. 文件描述符操作
这些函数操作使用文件描述符引用的 I/O stream。
文件描述符是与当前进程打开的文件相对应的一些小整数. 例如标准输入的通常文件描述符中 0，标准输出是 1,标准错误是 2. 进程打开的更多文件将被分配为 3，4，5,等. 这“文件描述符”有一点迷惑性;在 Unix 平台上，socket 和 pipe 通常也使用文件描述符引用.

os.close(fd)
关闭文件描述符 fd. 在 unix，Windows 中有效。

这函数是为低层的 I/O 服务的，应用在 os.open()或 pipe()返回的文件描述符上。关闭一个由内建函数 open()或 popen()或 fdopen()打开的文件对象，使用 close()方法。

os.closerange(fd_low, fd_high)

关闭从 fd_low（包含）到 fd_high（不包含）所有的文件描述符，忽略错误。在 unix，Windows 中有效。

等同于：

```
for fd in xrange(fd_low, fd_high):
    try:
        os.close(fd)
    except OSError:
        pass
```

os.dup(fd)

返回文件描述符 fd 的 cope．在 unix，Windows 中有效。

os.dup2(fd, fd2)

复制文件描述符 fd 到 fd2，如果有需要首先关闭 fd2．在 unix，Windows 中有效。

os.fchmod(fd, mode)

改变文件描述符为 fd 的文件’mode‘为 mode．查看 chmod()文档 中 mode 的值．在 unix 中有效，请查看相关文档.。

version 2.6 中新增.

os.fchown(fd, uid, gid)

改变文件描述符为 fd 的文件的所有者和 group 的 id 为 uid 和 gid．如果不想它们中的一个，就设置为-1．在 unix 中有效，请查看相关文档.。

version 2.6 中新增.

os.fdatasync(fd)

强制将文件描述符为 fd 的文件写入硬盘．不强制更新 metadata．在 unix 中有效,请查看相关文档.。

注：在 MacOS 中无效.

os.fpathconf(fd, name)

返回一个打开的文件的系统配置信息。name 为检索的系统配置的值，它也许是一个定义系统值的字符串，这些名字在很多标准中指定（POSIX.1，Unix 95，Unix 98，和其它）。一些平台也定义了一些额外的名字。这些名字在主操作系统上 pathconf_names 的字典中。对于不在 pathconf_names 中的配置变量，传递一个数字作为名字，也是可以接受的。 在 unix 中有效，请查看相关文档.。

如果 name 是一个字符串或者未知的，将抛出 ValueError。如果 name 是一个特别的值，在系统上不支持，即使它包含在 pathconf_names 中，将会抛出错误数字为 errno.EINVAL 的 OSError。

os.fstat(fd)
返回文件描述符 fd 的状态，像 stat()。在 unix，Windows 中有效。
os.fstatvfs(fd)
返回包含文件描述符 fd 的文件的文件系统的信息，像 statvfs(). 在 unix 中有效，请查看相关文档.。
os.fsync(fd)
强制将文件描述符为 fd 的文件写入硬盘.在 Unix，将调用 fsync()函数;在 Windows，调用 _commit()函数.

如果你准备操作一个 Python 文件对象 f，首先 f.flush(),然后 os.fsync(f.fileno()),确保与 f 相关的所有内存都写入了硬盘.在 unix，Windows 中有效。

os.ftruncate(fd, length)
裁剪文件描述符 fd 对应的文件，所以它最大不能超过文件大小. 在 unix 中有效，请查看相关文档.。
os.isatty(fd)
如果文件描述符 fd 是打开的,同时与 tty(-like)设备相连,则返回 true，否则 False. 在 unix 中有效，请查看相关文档.。
os.lseek(fd, pos, how)
设置文件描述符 fd 当前位置为 pos，how 方式修改: SEEK_SET 或者 0 设置从文件开始的计算的 pos; SEEK_CUR 或者 1 则从当前位置计算; os.SEEK_END 或者 2 则从文件尾部开始. 在 unix，Windows 中有效。
os.open(file, flags[, mode])
打开 file 同时根据 flags 设置变量 flags ,如果有 mode,则设置它的 mode. 默认的 mode 是 0777 (八进制)，当前掩码值是 first masked out. 返回刚打开的文件描述符. 在 unix，Windows 中有效。

flag 和 mode 值，请查看 C 运行时文档; flag 常数(像 O_RDONLY and O_WRONLY)在这个模块中也定义了（在下面）.

注:这函数是打算为低层 I/O 服务的.正常的使用，使用内建函数 open(),返回 read()和 write() 等方法创建的文件对象.包装文件描述符为“文件对象”，使用 fdopen().

os.openpty()
在一些 Unix 平台上有效，请查看相关文档.

os.pipe()
创建一个管道. 返回一对文件描述符(r, w) 分别为读和写. 在 unix，Windows 中有效。
os.read(fd, n)
从文件描述符 fd 中读取最多 n 个字节. 返回包含读取字节的 string. 文件描述符 fd 对应文件已达到结尾，返回一个空 string. 在 unix，Windows 中有效。

注:这函数是打算为低层 I/O 服务的 ，同时必须应用在 os.open()或者 pipe()函数返回的文件描述符．读取内建函数 open()或者 by popen()或者 fdopen(),或者 sys.stdin 返回的一个“文件对象” ，使用它的 read()或者 readline()方法．

os.tcgetpgrp(fd)
在 unix 中有效，请查看相关文档.。
os.tcsetpgrp(fd, pg)
在 unix 中有效，请查看相关文档.。
os.ttyname(fd)
在 unix 中有效，请查看相关文档.。
os.write(fd, str)
写入字符串到文件描述符 fd 中．返回实际写入的字符串长度．在 unix，Windows 中有效。

注:这函数是打算为低层 I/O 服务的 ，同时必须应用在 os.open()或者 pipe()函数返回的文件描述符．读取内建函数 open()或者 by popen()或者 fdopen(),或者 sys.stdin 返回的一个“文件对象” ，使用它的 read()或者 readline()方法．

下面的常数是 open()函数的 flags 参数选项．它们可以使用 bitwise 合并或者 operator |．它们中的一些常数并不是在所有平台都有效．它们更多使用请查看相关资料，在 unix 上参考 open(2) 手册页面，windows 上 http://msdn.microsoft.com/en-us/library/z0kc8e3z.aspx.

os.O_RDONLY
os.O_WRONLY
os.O_RDWR
os.O_APPEND
os.O_CREAT
os.O_EXCL
os.O_TRUNC
这些常数在 Unix and Windows 上有效.
os.O_DSYNC
os.O_RSYNC
os.O_SYNC
os.O_NDELAY
os.O_NONBLOCK
os.O_NOCTTY
os.O_SHLOCK
os.O_EXLOCK
这些常数仅在 Unix 上有效.
os.O_BINARY
os.O_NOINHERIT
os.O_SHORT_LIVED
os.O_TEMPORARY
os.O_RANDOM

os.O_SEQUENTIAL
os.O_TEXT
这些常数仅在 Windows 上有效.
os.O_ASYNC
os.O_DIRECT
os.O_DIRECTORY
os.O_NOFOLLOW
os.O_NOATIME
这些常数是 GNU 扩展，如果没有在 C 库声明刚没有.
os.SEEK_SET
os.SEEK_CUR
os.SEEK_END
lseek()函数的参数. 它们的值分别是 0, 1,和 2. 在 Unix and Windows 上有效.

版本 2.5 新增.

16.1.4. 文件和文件夹
os.access(path, mode)
使用现在的uid/gid尝试访问path. 注大部分操作使用有效的uid/gid, 因此运行环境可以在 suid/sgid 环境尝试，如果用户有权访问 path. mode 为 F_OK，测试存在的 path, 或者它可以是包含 R_OK, W_OK 和 X_OK 或者 R_OK, W_OK 和 X_OK 其中之一或者更多.如果允许访问返回 True , 否则返回 False. 查看 Unix 手册 access(2) 获取更多信息. 在 unix，Windows 中有效。

注:使用access()去测试用户是否授权. 在实际使用open()打开一个文件前测试会创建一个安全漏洞前 , 因为用户会利用这短暂时间在检测和打开这个文件去修改它.

注:即使 access()表明它将 succeed，但 I/O 操作也可能会失败, 如网络文件系统.

os.F_OK
作为 access()的 mode 参数，测试 path 是否存在.
os.R_OK
包含在 access()的 mode 参数中 ， 测试 path 是否可读.
os.W_OK
包含在 access()的 mode 参数中 ，测试 path 是否可写.
os.X_OK
包含在 access()的 mode 参数中 ，测试 path 是否可执行..
os.chdir(path)
改变当前工作目录. 在 unix，Windows 中有效。

os.fchdir(fd)
在 unix 中有效，请查看相关文档.。
os.getcwd()
返回当前工作目录的字符串， 在 unix，Windows 中有效。

os.getcwdu()
返回一个当前工作目录的 Unicode 对象. 在 unix，Windows 中有效。
os.chflags(path, flags)
在 unix 中有效，请查看相关文档.。

os.chroot(path)
在 unix 中有效，请查看相关文档.。

os.chmod(path, mode)
改变 path 的 mode 到数字 mode. mode 为下面中的一个 (在 stat 模块中定义)或者 bitwise 或者它们的组合:

?stat.S_ISUID
?stat.S_ISGID
?stat.S_ENFMT
?stat.S_ISVTX
?stat.S_IREAD
?stat.S_IWRITE
?stat.S_IEXEC
?stat.S_IRWXU
?stat.S_IRUSR
?stat.S_IWUSR
?stat.S_IXUSR
?stat.S_IRWXG
?stat.S_IRGRP
?stat.S_IWGRP
?stat.S_IXGRP
?stat.S_IRWXO
?stat.S_IROTH
?stat.S_IWOTH
?stat.S_IXOTH
在 unix，Windows 中有效。

注:尽管 Windows 支持 chmod(), 你只可以使用它设置只读 flag (通过 stat.S_IWRITE 和 stat.S_IREAD 常数或者一个相对应的整数)。所有其它的 bits 都忽略了.

os.chown(path, uid, gid)
在 unix 中有效，请查看相关文档.。
os.lchflags(path, flags)
在 unix 中有效，请查看相关文档.。

新增 version 2.6.

os.lchmod(path, mode)
在 unix 中有效，请查看相关文档.。

新增 version 2.6.

os.lchown(path, uid, gid)
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.link(source, link_name)
在 unix 中有效，请查看相关文档.。
os.listdir(path)
返回 path 指定的文件夹包含的文件或文件夹的名字的列表. 这个列表以字母顺序. 它不包括 '.' 和'..' 即使它在文件夹中. 在 unix，Windows 中有效。

Changed in version 2.3:在 Windows NT/2k/XP 和 Unix，如果文件夹是一个 Unicode object，结果将是 Unicode objects 列表. 不能解码的文件名将仍然作为 string objects 返回.

os.lstat(path)
像 stat(),但是没有符号链接. 这是 stat()的别名 在某些平台上,例如 Windows.
os.mkfifo(path[, mode])
在 unix 中有效，请查看相关文档.。

os.mknod(filename[, mode=0600, device])
创建一个名为 filename 文件系统节点（文件，设备特别文件或者命名 pipe）。 mode 指定创建或使用节点的权限，组合 (或者 bitwise) stat.S_IFREG, stat.S_IFCHR, stat.S_IFBLK, 和 stat.S_IFIFO (这些常数在 stat 模块). 对于 stat.S_IFCHR 和 stat.S_IFBLK, 设备定义了 最新创建的设备特殊文件 (可能使用 os.makedev()),其它都将忽略.

新增 version 2.3.

os.major(device)
从原始的设备号中提取设备 major 号码 (使用 stat 中的 st_dev 或者 st_rdev field).

新增 version 2.3.

os.minor(device)
从原始的设备号中提取设备 minor 号码 (使用 stat 中的 st_dev 或者 st_rdev field).

新增 version 2.3.

os.makedev(major, minor)
以 major 和 minor 设备号组成一个原始设备号.

新增 version 2.3.

os.mkdir(path[, mode])
以数字 mode 的 mode 创建一个名为 path 的文件夹.默认的 mode 是 0777 (八进制). 在有些平台上, mode 是忽略的. 当使用时。这当前的掩码值是 first masked out. 在 unix, Windows 中有效。

也可以创建临时文件夹; 查看 tempfile 模块 tempfile.mkdtemp()函数.

os.makedirs(path[, mode])
递归文件夹创建函数。像 mkdir(), 但创建的所有 intermediate-level 文件夹需要包含子文件夹. 抛出一个 error exception 如果子文件夹已经存在或者不能创建. 默认的 mode 是 0777 (八进制). 在有些平台上, mode 是忽略的.当使用时。这当前的掩码值是 first masked out。

注:
makedirs()变得迷惑 如果路径元素包含 os.pardir.
现在可以正确处理 UNC 路径.

os.pathconf(path, name)
在 unix 中有效，请查看相关文档.。

os.pathconf_names
在 unix 中有效，请查看相关文档.。
os.readlink(path)
在 unix 中有效，请查看相关文档.。

os.remove(path)
删除路径为 path 的文件.如果 path 是一个文件夹,将抛出 OSError; 查看下面的 rmdir()删除一个 directory. 这和下面的 unlink()函数文档是一样的.在 Windows,尝试删除一个正在使用的文件将抛出一个 exception;在 Unix,directory 入口会被删除,但分配给文件的存储是无效的，直到原来的文件不再使用. 在 unix, Windows 中有效。
os.removedirs(path)
递归删除 directorie. 像 rmdir(), 如果子文件夹成功删除, removedirs()才尝试它们的父文件夹,直到抛出一个 error(它基本上被忽略,因为它一般意味着你文件夹不为空). 例如, os.removedirs('foo/bar/baz') 将首先删除 'foo/bar/baz', 然后删除 'foo/bar' 和 'foo' 如果它们是空的. 如果子文件夹不能被成功删除,将抛出 OSError .

新增 version 1.5.2.

os.rename(src, dst)
重命名 file 或者 directory src 到 dst.如果 dst 是一个存在的 directory，将抛出 OSError. 在 Unix，如果 dst 在存且是一个 file,如果用户有权限的话，它将被安静的替换. 操作将会失败在某些 Unix 中如果 src 和 dst 在不同的文件系统中. 如果成功，这命名操作将会是一个原子操作 (这是 POSIX 需要). 在 Windows 上，如果 dst 已经存在，将抛出 OSError，即使它是一个文件. 在 unix，Windows 中有效。
os.renames(old, new)
递归重命名文件夹或者文件。像 rename()

新增 version 1.5.2.

os.rmdir(path)
删除 path 文件夹. 仅当这文件夹是空的才可以，否则，抛出 OSError. 要删除整个文件夹树，可以使用 shutil.rmtree(). 在 unix，Windows 中有效。
os.stat(path)
执行一个 stat()系统调用在给定的 path 上. 返回值是一个对象,属性与 stat 结构成员有关: st_mode (保护位), st_ino (inode number), st_dev (device), st_nlink (number of hard links), st_uid (所有用户的 id), st_gid (所有者 group id), st_size (文件大小，以位为单位), st_atime (最近访问的时间), st_mtime (最近修改的时间), st_ctime (依赖于平台;在 Unix 上是 metadata 最近改变的时间,或者在 Windows 上是创建时间):

```
>>> import os
>>> statinfo = os.stat('somefile.txt')
>>> statinfo
(33188, 422511L, 769L, 1, 1032, 100, 926L, 1105022698,1105022732,
1105022732)
>>> statinfo.st_size
926L
>>>
```

如果 stat_float_times()返回 True,time 值是 floats,以 second 进行计算. 一秒的小数部分也会显示出来， 如果系统支持. 在 Mac OS，时间常常是 floats.查看 stat_float_times() 获取更多信息.

在一些 Unix 系统上(例如 Linux)，下面的属性也许是有效的: st_blocks (为文件分配了多少块)，st_blksize (文件系统 blocksize)，st_rdev (设备型号如果是一个 inode 设备). st_flags (用户为文件定义的 flags).

在 unix，Windows 中有效。

os.stat_float_times([newvalue])
决定 stat_result 是否以 float 对象显示时间戳.

os.statvfs(path)
在 unix 中有效，请查看相关文档.。

os.symlink(source, link_name)
在 unix 中有效，请查看相关文档.。
os.tempnam([dir[, prefix]])
为创建一个临时文件返回一个唯一的 path。在 Windows 使用 TMP . 依赖于使用的 C 库;
警告:
使用 tempnam() 对于 symlink 攻击是一个漏洞; 考虑使用 tmpfile()代替.

在 unix，Windows 中有效。

os.tmpnam()
为创建一个临时文件返回一个唯一的 path.
Warning:
使用 tempnam() 对于 symlink 攻击是一个漏洞; 考虑使用 tmpfile()代替.

在 unix，Windows 中有效。

os.TMP_MAX
tmpnam() 将产生唯一名字的最大数值.
os.unlink(path)
删除 file 路径. 与 remove()相同; 在 unix，Windows 中有效。
os.utime(path, times)
返回指定的 path 文件的访问和修改的时间。如果时间是 None, 则文件的访问和修改设为当前时间 。 否则, 时间是一个 2-tuple 数字, (atime, mtime) 用来分别作为访问和修改的时间.
在 unix，Windows 中有效。

os.walk(top[, topdown=True[, onerror=None[, followlinks=False]]])
输出在文件夹中的文件名通过在树中游走，向上或者向下.在根目录下的每一个文件夹(包含它自己), 产生 3-tuple (dirpath, dirnames, filenames)【文件夹路径, 文件夹名字, 文件名】.

dirpath 是一个字符串, directory 的路径. dirnames 在 dirpath 中子文件夹的列表 (不包括 '.' '..'). filenames 文件是在 dirpath 不包含子文件夹的文件名的列表. 注：列表中的 names 不包含 path.为获得 dirpath 中的一个文件或者文件夹的完整路径 (以项目录开始)或者, 操作 os.path.join(dirpath, name).

如果 optional 参数 topdown 为 True 或者 not 指定, 一个 directory 的 3-tuple 将比它

的任何子文件夹的 3-tuple 先产生 (directories 自上而下).如果 topdown 为 False, 一个 directory 的 3-tuple 将比它的任何子文件夹的 3-tuple 后产生 (directories 自下而上)。

当 topdown 为 True,调用者可以修改列表中列出的文件夹名字(也可以使用 del 或者 slice), walk() 仅仅递归每一个包含在 dirnames 中的子文件夹; 可以减少查询, 利用访问的特殊顺序,或者甚至 告诉 walk() 关于文件夹的创建者或者重命名在它重新 walk() 之前.修改文件名当 topdown 为 False 时是无效的, 因为在 bottom-up 模式中在 dirnames 中的 directories 比 dirpath 它自己先产生 .

默认 listdir()的 errors 将被忽略. 如果 optional 参数 onerror 被指定,它应该是一个函数; 它调用时有一个参数, 一个 OSError 实例. 报告这错误后, 继续 walk,或者抛出 exception 终止 walk.注意 filename 是可用的的 , exception 对象的 filename 属性.

默认, walk() 不会进入符号链接 .

新增 version 2.6:

注:如果你传入一个相对的 pathname, 不要在 walk()执行过程中改变当前文件夹. walk() 不会改变改前文件夹, 同时确保它的调用者也不会改变.

```
import os
from os.path import join, getsize
for root, dirs, files in os.walk('python/Lib/email'):
    print root, "consumes",
    print sum(getsize(join(root, name)) for name in files),
    print "bytes in", len(files), "non-directory files"
    if 'CVS' in dirs:
        dirs.remove('CVS')  # don't visit CVS directories


# Delete everything reachable from the directory named in "top",
# assuming there are no symbolic links.
# CAUTION:  This is dangerous!  For example, if top == '/', it
# could delete all your disk files.
import os
for root, dirs, files in os.walk(top, topdown=False):
    for name in files:
        os.remove(os.path.join(root, name))
    for name in dirs:
```

```
        os.rmdir(os.path.join(root, name))
```
新增 version 2.3.

使用函数来创建和管理进程。
使用 exec*() 函数使用 arguments 列表来载入新程序. 在每个例子, 一个用户敲入一个命令行中的第一个参数传递给程序作为它自己的名字而不是作为参数. 对于 C 程序员来说, 它是传递给 main()的 argv[0]. 例如, os.execv('/bin/echo', ['foo', 'bar']) 将仅仅在标准输出上打印 bar; foo 将被忽略.

os.abort()
产生一个 SIGABRT 标识到当前的进程.在 Unix,这默认的行为是产生一个主要的 dump;在 Windows,这进程立即返回退出以一个状态码为3. 程序使用 signal.signal()来注册一个 SIGABRT 将导致不同的行为. 在 unix，Windows 中有效。
os.execl(path, arg0, arg1, ...)
os.execle(path, arg0, arg1, ..., env)
os.execlp(file, arg0, arg1, ...)
os.execlpe(file, arg0, arg1, ..., env)
os.execv(path, args)
os.execve(path, args, env)
os.execvp(file, args)
os.execvpe(file, args, env)
这些函数将执行一个新程序, 替换当前进程; 他们没有返回.在Unix,新的执行体载入到当前的进程, 同时将和当前的调用者有相同的 id. 将报告 Errors 当抛出 OSError 时.

当前的进程立即被替代. 打开文件对象和描述符不会被刷新, 如果在这些打开的文件中有数据缓冲区， 应该在调用 exec*() 函数之前， 使用 sys.stdout.flush()或者 os.fsync()flush 它们 .

在 unix，Windows 中有效。

os._exit(n)
使用状态 n 退出系统,没有调用清理函数，刷新缓冲区。 在 unix，Windows 中有效。

注:标准退出的方法是 sys.exit(n). _exit()一般使用于 fork()产生的子进程中.

os.EX_OK
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_USAGE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_DATAERR
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOINPUT
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOUSER
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOHOST
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_UNAVAILABLE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_SOFTWARE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_OSERR
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_OSFILE
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_CANTCREAT
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_IOERR
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_TEMPFAIL
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_PROTOCOL
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOPERM
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_CONFIG
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.EX_NOTFOUND
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.fork()
在 unix 中有效，请查看相关文档.。

os.forkpty()
在一些 unix 中有效，请查看相关文档
os.kill(pid, sig)
在 unix 中有效，请查看相关文档.。

os.killpg(pgid, sig)
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.nice(increment)
在 unix 中有效，请查看相关文档.。
os.plock(op)
在 unix 中有效，请查看相关文档.。
os.popen(...)
os.popen2(...)
os.popen3(...)
os.popen4(...)
运行子进程，返回交流的打开的管道.这些函数在前面创建文件对象时介绍过.
os.spawnl(mode, path, ...)
os.spawnle(mode, path, ..., env)
os.spawnlp(mode, file, ...)
os.spawnlpe(mode, file, ..., env)
os.spawnv(mode, path, args)
os.spawnve(mode, path, args, env)
os.spawnvp(mode, file, args)
os.spawnvpe(mode, file, args, env)
在新进程中执行程序 path
(请使用 subprocess 模块)

如果模式是 P_NOWAIT，返回新进程的 id;如果模式是 P_WAIT,返回进程退出时的状态码。如果正常退出,或者-signal，当 signal 是 killed.在 Windows，进程 id 实际上是 process 的 handle,所它可以使用于 waitpid()函数.

```
import os
os.spawnlp(os.P_WAIT, 'cp', 'cp', 'index.html', '/dev/null')

L = ['cp', 'index.html', '/dev/null']
os.spawnvpe(os.P_WAIT, 'cp', L, os.environ)
```

在 unix，Windows 中有效。

新增 version 1.6.

os.P_NOWAIT
os.P_NOWAITO
spawn*()族函数参数 mode 的可选值．如果给出其中任一个值，新进程一创建完成，spawn*()函数将立即返回 ,返回进程 id 的值．在 unix，Windows 中有效。

新增 version 1.6.

os.P_WAIT

spawn*()族函数参数 mode 的可能值. 如果将它赋值给 mode, spawn*() 函数不返回, 直接运行结束 以及如果运行成功 ,将返回进程的退出码,或者如果 signal 杀掉了这个进程,将返回-signal. 在 unix, Windows 中有效。

新增 version 1.6.

os.P_DETACH
os.P_OVERLAY
spawn*()族函数参数 mode 的可选值。P_DETACH 和 P_NOWAIT 很相似, 但是新进程依附在调用进程的 console 上。如果使用了 P_OVERLAY, 当前进程将被替换, spawn*()函数将无返回 。在 Windows 上有效。
新增 version 1.6.

os.startfile(path[, operation])
以相关的程序打开文件.

当 operation 没有指定或者'open', 这操作就像在 Windows Explorer 双击文件,或者将这个文件作为交互命令行中 start 命令的参数:与文件扩展相关的程序打开文件。

当指定另外操作时, 它必须是"command verb" 它指定应该对文件做什么.像 Microsoft 的'print' 'edit' (作用在文件上) 'explore' and 'find' (作用在文件夹上).

startfile()只要相关的应该程序一启动就返回。 没有选项等待应用程序关闭, 没有方法接收应用程序退出的状态. path 参数与当前 directory 相关. 如果你想使用一个绝对路径,确保第一个字符不是 ('/')。 使用 os.path.normpath() 函数确保路径对于 win32 来说是正确的. 在 Windows 上有效.

新增 version 2.0.

新增 version 2.5: The operation parameter.

os.system(command)
在 shell 中执行 string 命令. 这是使用标准 C 函数 system(), 有同样的限制. 例如改变 sys.stdin, 不影响命令执行环境.

在 Unix, 请查看相当文档.

在 Windows, 返回值是在 shell 运行命令的返回值。

在 unix, Windows 中有效。

subprocess 模块提供了一个更强大的功能产生新进程和接收它们的结果;

os.times()

返回一个 5-tuple 的浮点数字， 表示(处理器或者其它)累积时间，以秒为单位. items 为:用户时间，系统 time，子用户 time，子系统 time，和从过去一个固定的点真实流逝的时间. 在 unix，Windows 中有效。在 Windows，仅仅填充开始两项，其它都为 0.
os.wait()
在 unix 中有效，请查看相关文档.。
os.waitpid(pid, options)
Unix:等待一个指定的 pid 的子进程完成，返回一个 tuple 返回它的进程 id 和退出状态 . 一般情况下 option 设为 0.

更强细请查看相关文档

在 Windows: 等待一个指定的 pid 的进程完成，返回一个 tuple 返回它的进程 id 和退出状态向左移动了 8 位 。 如果 pid 小于或等于 0 没有特别的意思,将抛出 exception. integer options 没有任何影响. pid 可以指向任何进程的 id,不一定是子进程的 id.

os.wait3([options])
在 unix 中有效，请查看相关文档.。

新增 version 2.5.

os.wait4(pid, options)
在 unix 中有效，请查看相关文档.。

新增 version 2.5.

os.WNOHANG
在 unix 中有效，请查看相关文档.。
os.WCONTINUED
在某些 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.WUNTRACED
在 unix 中有效，请查看相关文档.。

os.WCOREDUMP(status)
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.WIFCONTINUED(status)
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.WIFSTOPPED(status)
在 unix 中有效，请查看相关文档.。
os.WIFSIGNALED(status)
在 unix 中有效，请查看相关文档.。
os.WIFEXITED(status)
在 unix 中有效，请查看相关文档.。
os.WEXITSTATUS(status)
在 unix 中有效，请查看相关文档.。
os.WSTOPSIG(status)
在 unix 中有效，请查看相关文档.。
os.WTERMSIG(status)
在 unix 中有效，请查看相关文档.。
16.1.6. Miscellaneous System Information
os.confstr(name)
在 unix 中有效，请查看相关文档.。

os.confstr_names
在 unix 中有效，请查看相关文档.。
os.getloadavg()
在 unix 中有效，请查看相关文档.。

新增 version 2.3.

os.sysconf(name)
在 unix 中有效，请查看相关文档.。

os.curdir
操作系统用此常数字符串作为当前文件夹的引用。
os.pardir
操作系统用此常数字符串作为父文件夹的引用。
os.sep
系统使用此字符来分割路径。
os.altsep
系统使用另外一个字符来分割路径，如果只有一个分割字符存在，则是 None.
os.extsep
分割基本文件名和扩展名的字符。

新增 version 2.2.

os.pathsep

系统使用此字符来分割搜索路径（像 PATH），例如 POSIX 上':',Windows 上的';'，也可以通过 os.path
os.defpath
默认的搜索路径用作 exec*p*()和 spawn*p*()如果环境没有 'PATH'. 也可以通过 os.path.
os.linesep
当前平台上的换行符字符串. 在 POSIX 上是'\n',或者 在 Windows 上是'\r\n' . 不要使用 os.linesep 作为换行符，当写入文本文件时 (默认); 使用'\n'代替, 在所有平台上.
os.devnull
空设备的文件路径.例如:POSIX 上 '/dev/null' . 也可以通过 os.path.

新增 version 2.4.

16.1.7. 其它 函数
os.urandom(n)
返回 n 个随机 byte 值的 string，作为加密使用

httplib – HTTP protocol client:
This module defines classes which implement the client side of the HTTP and HTTPS protocols. It is
normally not used directly – the module urllib uses it to handle URLs that use HTTP and HTTPS.
这是 python API 当中关于 httplib 的介绍，意思是:
这个组件实现了 HTTP 和 HTTPS 的客户端协议,它通常不被直接使用----urilib 使用它控制 URLs 用于 HTTP 和 HTTPS

urllib 和 urllib2:

urllib 和 urllib2 实现的功能大同小异，但 urllib2 要比 urllib 功能等各方面更高一个层次。目前的 HTTP 访问大部分都使用 urllib2 来进行访问

urllib2:

Python 代码

```
1. req = urllib2.Request('http://wwty.javaeye.com')
2. response = urllib2.urlopen(req)
3. the_page = response.read()
```

FTP 同样:

Python 代码

```
1. req = urllib2.Request('ftp://wwty.javaeye.com')
```

urlopen 返回的应答对象 response 有两个很有用的方法 info()和 geturl()
geturl -- 这个返回获取的真实的 URL，这个很有用，因为 urlopen(或者 opener 对象使用的)或许
会有重定向。获取的 URL 或许跟请求 URL 不同。

Data 数据
有时候你希望发送一些数据到 URL
post 方式：

Python 代码

```
1. values ={'body' : 'test short talk','via':'xxxx'}
2. data = urllib.urlencode(values)
3. req = urllib2.Request(url, data)
```

get 方式：

Python 代码

```
1. data['name'] = 'Somebody Here'
2. data['location'] = 'Northampton'
3. data['language'] = 'Python'
4. url_values = urllib.urlencode(data)
5. url = 'http://www.example.com/example.cgi'
6. full_url = url + '?' + url_values
7. data = urllib2.open(full_url)
```

URLError--HTTPError:
Python 代码

```
1. from urllib2 import Request, urlopen, URLError, HTTPError
2. req = Request(someurl)
3. try:
4.       response = urlopen(req)
5. except HTTPError, e:
6.      print 'Error code: ', e.code
7. except URLError, e:
8.      print 'Reason: ', e.reason
9. else:
10.        ..............
```

或者：

Python 代码

```
1. from urllib2 import Request, urlopen, URLError
2. req = Request(someurl)
3. try:
4.       response = urlopen(req)
5. except URLError, e:
6.      if hasattr(e, 'reason'):
7.            print 'Reason: ', e.reason
8.      elif hasattr(e, 'code'):
9.            print 'Error code: ', e.code
10.      else:
11.             .............
```

通常，URLError 在没有网络连接(没有路由到特定服务器),或者服务器不存在的情况下产生
异常同样会带有"reason"属性，它是一个 tuple，包含了一个错误号和一个错误信息

```
req = urllib2.Request('http://www.pretend_server.org')
try:
   urllib2.urlopen(req)
except URLError, e:
   print e.reason
   print e.code
   print e.read()
```

最后需要注意的就是，当处理 URLError 和 HTTPError 的时候，应先处理 HTTPError，后处理 URLError

Openers 和 Handlers：
opener 使用操作器（handlers）。所有的重活都交给这些 handlers 来做。每一个 handler 知道
怎么打开 url 以一种独特的 url 协议（http，ftp 等等），或者怎么处理打开 url 的某些方面，如，HTTP 重定向，或者 HTTP
cookie。
默认 opener 有对普通情况的操作器 （handlers）- ProxyHandler, UnknownHandler, HTTPHandler, HTTPDefaultErrorHandler, HTTPRedirectHandler, FTPHandler, FileHandler, HTTPErrorProcessor.
再看 python API：Return an OpenerDirector instance, which chains the handlers in the order given
这就更加证明了一点，那就是 opener 可以看成是一个容器，这个容器当中放的是控制器，默认的，容器当中有五个控制
器，程序员也可以加入自己的控制器，然后这些控制器去干活。

Python 代码

```
1. class HTTPHandler(AbstractHTTPHandler):
2.     def http_open(self, req):
3.         return self.do_open(httplib.HTTPConnection,
   req)
4.      http_request = AbstractHTTPHandler.do_request_
```

HTTPHandler 是 Openers 当中的默认控制器之一，看到这个代码，证实了 urllib2 是借助于 httplib 实现的，同时也证实了 Openers 和 Handlers 的关系。

初学 Python，对 Python 的文字处理能力有很深的印象，除了 str 对象自带的一些方法外，就是正则表达式这个强大的模块了。但是对于初学者来说，要用好这个功能还是有点难度，我花了好长时间才摸出了点门道。由于我记性不好，很容易就忘事，所以还是写下来比较好一些，同时也可以加深印象，整理思路。

由于我是初学，所以肯定会有些错误，还望高手不吝赐教，指出我的错误。

1 Python 正则式的基本用法

Python 的正则表达式的模块是 ‘re’,它的基本语法规则就是指定一个字符序列，比如你要在一个字符串 s=’123abc456’ 中查找字符串 ’abc’,只要这样写：

```
>>> import re
>>> s='123abc456eabc789'
>>> re.findall(r’abc’,s)
```

结果就是：

```
['abc', 'abc']
```

这里用到的函数 ”findall(rule , target [,flag])” 是个比较直观的函数，就是在目标字符串中查找符合规则的字符串。第一个参数是规则，第二个参数是目标字符串，后面还可以跟一个规则选项（选项功能将在 compile 函数的说明中详细说明）。返回结果结果是一个**列表**，中间存放的是符合规则的字符串。如果没有符合规则的字符串被找到，就返回一个**空**列表。

为什么要用 r’ ..‘字符串（raw 字符串）？ 由于正则式的规则也是由一个字符串定义的，而在正则式中大量使用转义字符’\’，如果不用 raw 字符串，则在需要写一个’\’的地方，你必须得写成’\\’,那么在要从目标字符串中匹配一个’\’的时候，你就得写上 4 个’\’成为’\\\\’！这当然很麻烦，也不直观，所以一般都使用 r’’来定义规则字符串。当然，某些情况下，可能不用 raw 字符串比较好。

以上是个最简单的例子。当然实际中这么简单的用法几乎没有意义。为了实现复杂的规则查找，re 规定了若干语法规则。它们分为这么几类：

功能字符 ： ‘.’ ‘*’ ‘+’ ‘|’ ‘?’ ‘^’ ‘$’ ‘\’ 等，它们有特殊的功能含义。特别是’\’字符，它是转义引导符号，跟在它后面的字符一般有特殊的含义。

规则分界符： ‘[‘ ‘]’ ‘（’ ‘）’ ‘{‘ ‘}’ 等，也就是几种括号了。

预定义转义字符集： “\d” “\w” “\s” 等等，它们是以字符’\’开头，后面接一个特定字符的形式，用来指示一个预定义好的含义。

其它特殊功能字符： ’#’ ‘!’ ‘:’ ‘-‘等，它们只在特定的情况下表示特殊的含义，比如(?# …)就表示一个注释，里面的内容会被忽略。

下面来一个一个的说明这些规则的含义，不过说明的顺序并不是按照上面的顺序来的，而是我认为由浅入深，由基本到复杂的顺序来编排的。同时为了直观，在说明的过程中尽量多举些例子以方便理解。

1.1 基本规则

‘[‘ ‘]’ 字符集合设定符

首先说明一下字符集合设定的方法。由一对方括号括起来的字符，表明一个字符集合，能够匹配包含在其中的任意一个字符。比如 [abc123]，表明字符’a’ ‘b’ ‘c’ ‘1’ ‘2’ ‘3’都符合它的要求。可以被匹配。

在’[‘ ‘]’中还可以通过 ’-‘ 减号来指定一个字符集合的范围，比如可以用[a-zA-Z]来指定所以英文字母的大小写，因为英文字母是按照从小到大的顺序来排的。你不可以把大小的顺序颠倒了，比如写成[z-a]就不对了。

如果在’[‘ ‘]’里面的开头写一个 ‘^’ 号，则表示取非，即在括号里的字符都不匹配。如[^a-zA-Z]表明不匹配所有英文字母。但是如果 ‘^’不在开头，则它就不再是表示取非，而表示其本身，如[a-z^A-Z]表明匹配所有的英文字母和字符’^’。

‘|’或规则

将两个规则并列起来，以‘|’连接，表示只要满足其中之一就可以匹配。比如

[a-zA-Z]|[0-9] 表示满足数字或字母就可以匹配，这个规则等价于 [a-zA-Z0-9]

注意：关于’|’要注意两点：

第一，它在’[‘ ‘]’之中不再表示或，而表示他本身的字符。如果要在’[‘ ‘]’外面表示一个’|’字符，必须用反斜杠引导，即 ’\|’ ；

第二，它的有效范围是它两边的整条规则，比如‘dog|cat’匹配的是‘dog’和’cat’，而不是’g’和’c’。如果想限定它的有效范围，必需使用一个无捕获组 ‘(?:)’包起来。比如要匹配 ‘I have a dog’或’I have a cat’，需要写成 r’I have a (?:dog|cat)’ ，而不能写成 r’I have a dog|cat’

例

```
>>> s = ‘I have a dog , I have a cat’
>>> re.findall( r’I have a (?:dog|cat)’ , s )
['I have a dog', 'I have a cat']        #正如我们所要的
```

下面再看看不用无捕获组会是什么后果：

```
>>> re.findall( r’I have a dog|cat’ , s )
['I have a dog', 'cat']                 #它将’I have a dog’和’cat’当成两个规则了
```

至于无捕获组的使用，后面将仔细说明。这里先跳过。

‘.’匹配所有字符

匹配除换行符’\n’外的所有字符。如果使用了’S’选项，匹配包括’\n’的所有字符。

例：

```
>>> s=’123 \n456 \n789’
>>> findall(r‘.+’,s)
['123', '456', '789']
>>> re.findall(r‘.+’ , s , re.S)
['123\n456\n789']
```

‘^’和’$’ 匹配字符串开头和结尾

注意’^’不能在‘[ ]’中，否则含意就发生变化，具体请看上面的’[‘ ‘]’说明。 在多行模式下，它们可以匹配每一行的行首和行尾。具体请看后面 compile 函数说明的’M’选项部分

‘\d’ 匹配数字

这是一个以’\’开头的转义字符，’\d’表示匹配一个数字，即等价于[0-9]

‘\D’ 匹配非数字

这个是上面的反集，即匹配一个非数字的字符，等价于[^0-9]。注意它们的大小写。下面我们还将看到 Python 的正则规则中很多转义字符的大小写形式，代表互补的关系。这样很好记。

‘\w’ 匹配字母和数字

匹配所有的英文字母和数字，即等价于[a-zA-Z0-9]。

‘\W’ 匹配非英文字母和数字

即’\w’的补集，等价于[^a-zA-Z0-9]。

‘\s’ 匹配间隔符

即匹配空格符、制表符、回车符等表示分隔意义的字符，它等价于[\t\r\n\f\v]。（注意最前面有个空格)

‘\S’ 匹配非间隔符

即间隔符的补集，等价于[^ \t\r\n\f\v]

‘\A’ 匹配字符串开头

匹配字符串的开头。它和’^’的区别是，’\A’只匹配整个字符串的开头，即使在’M’模式下，它也不会匹配其它行的很首。

‘\Z’ 匹配字符串结尾

匹配字符串的结尾。它和’$’的区别是，’\Z’只匹配整个字符串的结尾，即使在’M’模式下，它也不会匹配其它各行的行尾。

例：

```
>>> s= '12 34\n56 78\n90'
>>> re.findall( r'^\d+' , s , re.M )       #匹配位于行首的数字
['12', '56', '90']
>>> re.findall( r’\A\d+’, s , re.M )       #匹配位于字符串开头的数字
['12']
>>> re.findall( r'\d+$' , s , re.M )       #匹配位于行尾的数字
['34', '78', '90']
>>> re.findall( r’\d+\Z’ , s , re.M )      #匹配位于字符串尾的数字
['90']
```

‘\b’ 匹配单词边界

它匹配一个单词的边界，比如空格等，不过它是一个‘0’长度字符，它匹配完的字符串不会包括那个分界的字符。而如果用’\s’来匹配的话，则匹配出的字符串中会包含那个分界符。

例：

```
>>> s =  'abc abcde bc bcd'
>>> re.findall( r’\bbc\b’ , s )      #匹配一个单独的单词 ‘bc’ ，而当它是其它单词的一部分的时候不匹配
['bc']                              ＃只找到了那个单独的’bc’
>>> re.findall( r’\sbc\s’ , s )      ＃匹配一个单独的单词 ‘bc’
[' bc ']                            #只找到那个单独的’bc’，不过注意前后有两个空格，可能有点看不清楚
```

‘\B’ 匹配非边界

和’\b’相反，它只匹配非边界的字符。它同样是个 0 长度字符。

接上例：

```
>>> re.findall( r’\Bbc\w+’ , s )    #匹配包含’bc’但不以’bc’为开头的单词
['bcde']                            #成功匹配了’abcde’中的’bcde’，而没有匹配’bcd’
```

‘(?:)’ 无捕获组

当你要将一部分规则作为一个整体对它进行某些操作，比如指定其重复次数时，你需要将这部分规则用’(?:’ ‘)’把它包围起来，而不能仅仅只用一对括号，那样将得到绝对出人意料的结果。

例：匹配字符串中重复的’ab’

```
>>> s=’ababab abbabb aabaab’
>>> re.findall( r’\b(?:ab)+\b’ , s )
['ababab']
```

如果仅使用一对括号，看看会是什么结果：

```
>>> re.findall( r'\b(ab)+\b' , s )
['ab']
```

这是因为如果只使用一对括号，那么这就成为了一个组(group)。组的使用比较复杂，将在后面详细讲解。

'(?#)' 注释

Python 允许你在正则表达式中写入注释，在'(?#' ')'之间的内容将被忽略。

(?iLmsux) 编译选项指定

Python 的正则式可以指定一些选项，这个选项可以写在 findall 或 compile 的参数中，也可以写在正则式里，成为正则式的一部分。这在某些情况下会便利一些。具体的选项含义请看后面的 compile 函数的说明。

此处编译选项'i' 等价于 IGNORECASE ，L 等价于 LOCAL ，m 等价于 MULTILINE ，s 等价于 DOTALL ，u 等价于 UNICODE ， x 等价于 VERBOSE 。

请注意它们的大小写。在使用时可以只指定一部分，比如只指定忽略大小写，可写为 '(?i)'，要同时忽略大小写并使用多行模式，可以写为 '(?im)'。

另外要注意选项的有效范围是整条规则，即写在规则的任何地方，选项都会对全部整条正则式有效。

1.2 重复

正则式需要匹配不定长的字符串，那就一定需要表示重复的指示符。Python 的正则式表示重复的功能很丰富灵活。重复规则的一般的形式是在一条字符规则后面紧跟一个表示重复次数的规则，已表明需要重复前面的规则一定的次数。重复规则有：

'*' 0 或多次匹配

表示匹配前面的规则 0 次或多次。

‘+’1 次或多次匹配

表示匹配前面的规则至少 1 次，可以多次匹配

例：匹配以下字符串中的前一部分是字母，后一部分是数字或没有的变量名字

```
>>> s = ‘ aaa bbb111 cc22cc 33dd ‘
>>> re.findall( r’\b[a-z]+\d*\b’ , s )    #必须至少 1 个字母开头，以连续数字结尾或没有数字
['aaa', 'bbb111']
```

注意上例中规则前后加了表示单词边界的’\b’指示符，如果不加的话结果就会变成：

```
>>> re.findall( r’[a-z]+\d*’ , s )
['aaa', 'bbb111', 'cc22', 'cc', 'dd']  #把单词给拆开了
```

大多数情况下这不是我们期望的结果。

‘?’0 或 1 次匹配

只匹配前面的规则 0 次或 1 次。

例，匹配一个数字，这个数字可以是一个整数，也可以是一个科学计数法记录的数字，比如 123 和 10e3 都是正确的数字。

```
>>> s = ‘ 123 10e3 20e4e4 30ee5 ‘
>>> re.findall( r’ \b\d+[eE]?\d*\b’ , s )
['123', '10e3']
```

它正确匹配了 123 和 10e3,正是我们期望的。注意前后的’\b’的使用，否则将得到不期望的结果。

1.2.1 精确匹配和最小匹配

Python 正则式还可以精确指定匹配的次数。指定的方式是

‘{m}’ 精确匹配 m 次

‘{m,n}’ 匹配最少 m 次，最多 n 次。(n>m)

如果你只想指定一个最少次数或只指定一个最多次数，你可以把另外一个参数空起来。比如你想指定最少 3 次，可以写成 {3,}（注意那个逗号），同样如果只想指定最大为 5 次，可以写成{，5}，也可以写成{0,5}。

例 寻找下面字符串中

a：3 位数

b：2 位数到 4 位数

c：5 位数以上的数

d：4 位数以下的数

```
>>> s= ‘ 1 22 333 4444 55555 666666 ‘
>>> re.findall( r’\b\d{3}\b’ , s )     # a：3 位数
['333']
>>> re.findall( r’\b\d{2,4}\b’ , s )      # b：2 位数到 4 位数
['22', '333', '4444']
>>> re.findall( r’\b\d{5,}\b’, s )     # c：5 位数以上的数
['55555', '666666']
>>> re.findall( r’\b\d{1,4}\b’ , s )      # 4 位数以下的数
['1', '22', '333', '4444']
```

‘*?’ ‘+?’ ‘??’ 最小匹配

‘*’ ‘+’ ‘?’通常都是尽可能多的匹配字符。有时候我们希望它尽可能少的匹配。比如一个 c 语言的注释 ‘/* part 1 */ /* part 2 */’，如果使用最大规则：

```
>>> s =r ‘/* part 1 */ code /* part 2 */’
>>> re.findall( r’/\*.*\*/’ , s )
[‘/* part 1 */ code /* part 2 */’]
```

结果把整个字符串都包括进去了。如果把规则改写成

```
>>> re.findall( r’/\*.*?\*/’ , s )          #在*后面加上?，表示尽可能少的匹配
['/* part 1 */', '/* part 2 */']
```

结果正确的匹配出了注释里的内容

1.3前向界定与后向界定

有时候需要匹配一个跟在特定内容后面的或者在特定内容前面的字符串，Python 提供一个简便的前向界定和后向界定功能，或者叫前导指定和跟从指定功能。它们是：

‘(?<=…)’ 前向界定

括号中’…’代表你希望匹配的字符串的前面应该出现的字符串。

‘(?=…)’ 后向界定

括号中的’…’代表你希望匹配的字符串后面应该出现的字符串。

例： 你希望找出 c 语言的注释中的内容，它们是包含在’/*’和’*/’之间，不过你并不希望匹配的结果把’/*’和’*/’也包括进来，那么你可以这样用：

```
>>> s=r’/* comment 1 */  code  /* comment 2 */’
>>> re.findall( r’(?<=/\*).+?(?=\*/)’ , s )
[' comment 1 ', ' comment 2 ']
```

注意这里我们仍然使用了最小匹配，以避免把整个字符串给匹配进去了。

要注意的是，前向界定括号中的表达式必须是常值，也即你不可以在前向界定的括号里写正则式。比如你如果在下面的字符串中想找到被字母夹在中间的数字，你不可以用前向界定：

例：

```
>>> s = ‘aaa111aaa , bbb222 , 333ccc ‘
>>> re.findall( r’(?<=[a-z]+)\d+(?=[a-z]+)' , s )      # 错误的用法
```

它会给出一个错误信息：

```
error: look-behind requires fixed-width pattern
```

不过如果你只要找出后面接着有字母的数字，你可以在后向界定写正则式：

```
>>> re.findall( r'\d+(?=[a-z]+)', s )
['111', '333']
```

如果你一定要匹配包夹在字母中间的数字，你可以使用组（group）的方式

```
>>> re.findall (r'[a-z]+(\d+)[a-z]+' , s )
['111']
```

组的使用将在后面详细讲解。

除了前向界定前向界定和后向界定外，还有前向非界定和后向非界定，它的写法为：

‘(?<!...)’前向非界定

只有当你希望的字符串前面不是’…’的内容时才匹配

‘(?!...)’后向非界定

只有当你希望的字符串后面不跟着’…’内容时才匹配。

接上例，希望匹配后面不跟着字母的数字

```
>>> re.findall( r'\d+(?!\w+)' , s )
['222']
```

注意这里我们使用了\w 而不是像上面那样用[a-z]，因为如果这样写的话，结果会是：

```
>>> re.findall( r'\d+(?![a-z]+)' , s )
['11', '222', '33']
```

这和我们期望的似乎有点不一样。它的原因，是因为’111’和’222’中的前两个数字也是满足这个要求的。因此可看出，正则式的使用还是要相当小心的，因为我开始就是这样写的，看到结果后才明白过来。不过 Python 试验起来很方

便，这也是脚本语言的一大优点，可以一步一步的试验，快速得到结果，而不用经过烦琐的编译、链接过程。也因此学习 Python 就要多试，跌跌撞撞的走过来，虽然曲折，却也很有乐趣。

1.4 组的基本知识

上面我们已经看过了 Python 的正则式的很多基本用法。不过如果仅仅是上面那些规则的话，还是有很多情况下会非常麻烦，比如上面在讲前向界定和后向界定时，取夹在字母中间的数字的例子。用前面讲过的规则都很难达到目的，但是用了组以后就很简单了。

‘(‘’)’无命名组

最基本的组是由一对圆括号括起来的正则式。比如上面匹配包夹在字母中间的数字的例子中使用的(\d+)，我们再回顾一下这个例子：

```
>>> s = ‘aaa111aaa , bbb222 , 333ccc ‘
>>> re.findall (r'[a-z]+(\d+)[a-z]+' , s )
['111']
```

可以看到 findall 函数只返回了包含在’()’中的内容，而虽然前面和后面的内容都匹配成功了，却并不包含在结果中。

除了最基本的形式外，我们还可以给组起个名字，它的形式是

‘(?P<name>…)’ 命名组

‘(?P’代表这是一个 Python 的语法扩展’<…>’里面是你给这个组起的名字，比如你可以给一个全部由数字组成的组叫做’num’，它的形式就是’(?P<num>\d+)’。起了名字之后，我们就可以在后面的正则式中通过名字调用这个组，它的形式是

‘(?P=name)’ 调用已匹配的命名组

要注意，再次调用的这个组是已被匹配的组，也就是说它里面的内容是和前面命名组里的内容是一样的。

我们可以看更多的例子：请注意下面这个字符串各子串的特点。

```
>>> s='aaa111aaa,bbb222,333ccc,444ddd444,555eee666,fff777ggg'
```

我们看看下面的正则式会返回什么样的结果：

```
>>> re.findall( r'([a-z]+)\d+([a-z]+)' , s )     # 找出中间夹有数字的字母
[('aaa', 'aaa'), ('fff', 'ggg')]
>>> re.findall( r '(?P<g1>[a-z]+)\d+(?P=g1)' , s )  #找出被中间夹有数字的前后同样的字母
['aaa']
>>> re.findall( r'[a-z]+(\d+)([a-z]+)' , s )     #找出前面有字母引导，中间是数字，后面是字母的字符串中的中间的数字和后面的字母
[('111', 'aaa'), ('777', 'ggg')]
```

我们可以通过命名组的名字在后面调用已匹配的命名组，不过名字也不是必需的。

‘\number’　　通过序号调用已匹配的组

正则式中的每个组都有一个序号，序号是按组从左到右，从 1 开始的数字，你可以通过下面的形式来调用已匹配的组

比如上面找出被中间夹有数字的前后同样的字母的例子，也可以写成：

```
>>> re.findall( r’([a-z]+)\d+\1’ , s )
['aaa']
```

结果是一样的。

我们再看一个例子

```
>>> s='111aaa222aaa111 , 333bbb444bb33'
>>> re.findall( r'(\d+)([a-z]+)(\d+)(\2)(\1)' , s )    #找出完全对称的 数字－字母－数字－字母－数字 中的数字和字母
[('111', 'aaa', '222', 'aaa', '111')]
```

Python2.4 以后的 re 模块，还加入了一个新的条件匹配功能

‘(?(*id/name*)yes-pattern|no-pattern)’ 判断指定组是否已匹配，执行相应的规则

这个规则的含义是，如果 id/name 指定的组在前面匹配成功了，则执行 yes-pattern 的正则式，否则执行 no-pattern 的正则式。

举个例子，比如要匹配一些形如 usr@mail 的邮箱地址，不过有的写成< usr@mail >即用一对<>括起来，有点则没有，要匹配这两种情况，可以这样写

```
>>> s='<usr1@mail1>  usr2@maill2'
>>> re.findall( r'(<)?\s*(\w+@\w+)\s*(?(1)>)' , s )
[('<', 'usr1@mail1'), ('', 'usr2@maill2')]
```

不过如果目标字符串如下

```
>>> s='<usr1@mail1>  usr2@maill2 <usr3@mail3   usr4@mail4>
< usr5@mail5 '
```

而你想得到要么由一对<>包围起来的一个邮件地址，要么得到一个没有被<>包围起来的地址，但不想得到一对<>中间包围的多个地址或不完整的<>中的地址，那么使用这个式子并不能得到你想要的结果

```
>>> re.findall( r'(<)?\s*(\w+@\w+)\s*(?(1)>)' , s )
[('<', 'usr1@mail1'), ('', 'usr2@maill2'), ('', 'usr3@mail3
'), ('', 'usr4@mail4'), ('', 'usr5@mail5')]
```

它仍然找到了所有的邮件地址。

想要实现这个功能，单纯的使用 findall 有点吃力，需要使用其它的一些函数，比如 match 或 search 函数，再配合一些控制功能。这部分的内容将在下面详细讲解。

小结：以上基本上讲述了 Python 正则式的语法规则。虽然大部分语法规则看上去都很简单，可是稍不注意，仍然会得到与期望大相径庭的结果，所以要写好正则式，需要仔细的体会正则式规则的含义后不同规则之间细微的差别。

详细的了解了规则后，再配合后面就要介绍的功能函数，就能最大的发挥正则式的威力了。

2 re 模块的基本函数

在上面的说明中，我们已经对 re 模块的基本函数 ‘findall’很熟悉了。当然如果光有 findall 的话，很多功能是不能实现的。下面开始介绍一下 re 模块其它的常用基本函数。灵活搭配使用这些函数，才能充分发挥 Python 正则式的强大功能。

首先还是说下老熟人 findall 函数吧

findall(rule , target [,flag])

在目标字符串中查找符合规则的字符串。

第一个参数是规则，第二个参数是目标字符串，后面还可以跟一个规则选项（选项功能将在 compile 函数的说明中详细说明）。

返回结果结果是一个**列表**，中间存放的是符合规则的字符串。如果没有符合规则的字符串被找到，就返回一个**空**列表。

2.1 使用 compile 加速

compile(rule [,flag])

将正则规则编译成一个 Pattern 对象，以供接下来使用。

第一个参数是规则式，第二个参数是规则选项。

返回一个 Pattern 对象

直接使用 findall (rule , target)的方式来匹配字符串，一次两次没什么，如果是多次使用的话，由于正则引擎每次都要把规则解释一遍，而规则的解释又是相当费时间的，所以这样的效率就很低了。如果要多次使用同一规则来进行匹配的话，可以使用 re.compile 函数来将规则预编译，使用编译过返回的 Regular Expression Object 或叫做 Pattern 对象来进行查找。

例

```
>>> s='111,222,aaa,bbb,ccc333,444ddd'
>>> rule=r’\b\d+\b’
>>> compiled_rule=re.compile(rule)
```

```
>>> compiled_rule.findall(s)
['111', '222']
```

可见使用 compile 过的规则使用和未编译的使用很相似。compile 函数还可以指定一些规则标志，来指定一些特殊选项。多个选项之间用 ’|’（位或）连接起来。

I　IGNORECASE 忽略大小写区别。

L　LOCAL 字符集本地化。这个功能是为了支持多语言版本的字符集使用环境的，比如在转义符\w，在英文环境下，它代表[a-zA-Z0-9]，即所以英文字符和数字。如果在一个法语环境下使用，缺省设置下，不能匹配"é" 或 "ç"。加上这 L 选项和就可以匹配了。不过这个对于中文环境似乎没有什么用，它仍然不能匹配中文字符。

M　MULTILINE 多行匹配。在这个模式下’^’(代表字符串开头)和’$’(代表字符串结尾)将能够匹配多行的情况，成为行首和行尾标记。比如

```
>>> s=’123 456\n789 012\n345 678’
>>> rc=re.compile(r’^\d+’)   #匹配一个位于开头的数字，没有使用 M 选项
>>> rc.findall(s)
['123']      #结果只能找到位于第一个行首的’123’
>>> rcm=re.compile(r’^\d+’,re.M)   #使用 M 选项
>>> rcm.findall(s)
['123', '789', '345'] #找到了三个行首的数字
```

同样，对于’$’来说，没有使用 M 选项，它将匹配最后一个行尾的数字，即’678’，加上以后，就能匹配三个行尾的数字 456 012 和 678 了.

```
>>> rc=re.compile(r’\d+$’)
>>> rcm=re.compile(r’\d+$’,re.M)
>>> rc.findall(s)
```

```
['678']
>>> rcm.findall(s)
['456', '012', '678']
```

S DOTALL **‘.’**号将匹配所有的字符。缺省情况下**’.’**匹配除换行符**’\n’**外的所有字符，使用这一选项以后，**’.’**就能匹配包括**’\n’**的任何字符了。

U UNICODE \w, \W, \b, \B, \d, \D, \s 和 \S 都将使用 Unicode。

X VERBOSE 这个选项忽略规则表达式中的空白，并允许使用**’#’**来引导一个注释。这样可以让你把规则写得更美观些。比如你可以把规则

```
>>> rc = re.compile(r"\d+|[a-zA-Z]+")   #匹配一个数字或者单词
```

使用 X 选项写成：

```
>>> rc = re.compile(r"""# start a rule
    \d+                 # number
    | [a-zA-Z]+         # word
    """, re.VERBOSE)
```

在这个模式下，如果你想匹配一个空格，你必须用'\ '的形式（'\'后面跟一个空格）

2.2 match 与 search

match(rule , targetString [,flag])

search(rule , targetString [,flag])

（注：re 的 match 与 search 函数同 compile 过的 Pattern 对象的 match 与 search 函数的参数是不一样的。Pattern 对象的 match 与 search 函数更为强大，是真正最常用的函数）

按照规则在目标字符串中进行匹配。

第一个参数是正则规则，第二个是目标字符串，第三个是选项（同 compile 函数的选项）

返回：若成功返回一个 Match 对象，失败无返回

findall 虽然很直观，但是在进行更复杂的操作时，就有些力不从心了。此时更多的使用的是 match 和 search 函数。他们的参数和 findall 是一样的，都是：

```
match( rule , targetString [,flag] )
search( rule , targetString [,flag] )
```

不过它们的返回不是一个简单的字符串列表，而是一个 MatchObject （如果匹配成功的话）.。通过操作这个 matchObject，我们可以得到更多的信息。

需要注意的是，如果匹配不成功，它们则返回一个 NoneType。所以在对匹配完的结果进行操作之前，你必需先判断一下是否匹配成功了，比如：

```
>>> m=re.match( rule , target )
>>> if m:            #必需先判断是否成功
    doSomethin
```

这两个函数唯一的区别是：match 从字符串的开头开始匹配，如果开头位置没有匹配成功，就算失败了；而 search 会跳过开头，继续向后寻找是否有匹配的字符串。针对不同的需要，可以灵活使用这两个函数。

关于 match 返回的 MatchObject 如果使用的问题，是 Python 正则式的精髓所在，它与组的使用密切相关。我将在下一部分详细讲解，这里只举个最简单的例子：

例：

```
>>> s= 'Tom:9527 , Sharry:0003'
>>> m=re.match( r'(?P<name>\w+):(?P<num>\d+)' , s )
>>> m.group()
'Tom:9527'
>>> m.groups()
('Tom', '9527')
```

```
>>> m.group(‘name’)
'Tom'
>>> m.group(‘num’)
'9527'
```

2.3 finditer

finditer(rule , target [,flag])

参数同 findall

返回一个迭代器

finditer 函数和 findall 函数的区别是，findall 返回所有匹配的字符串，并存为一个列表，而 finditer 则并不直接返回这些字符串，而是返回一个迭代器。关于迭代器，解释起来有点复杂，还是看看例子把：

```
>>> s=’111 222 333 444’
>>> for i in re.finditer(r’\d+’ , s ):
    print i.group(),i.span()      #打印每次得到的字符串和起始结束位置
```

结果是

```
111 (0, 3)
222 (4, 7)
333 (8, 11)
444 (12, 15)
```

简单的说吧，就是 finditer 返回了一个可调用的对象，使用 for i in finditer()的形式，可以一个一个的得到匹配返回的 Match 对象。这在对每次返回的对象进行比较复杂的操作时比较有用。

2.4 字符串的替换和修改

re 模块还提供了对字符串的替换和修改函数，他们比字符串对象提供的函数功能要强大一些。这几个函数是

sub (rule , replace , target [,count])

subn(rule , replace , target [,count])

在目标字符串中规格规则查找匹配的字符串，再把它们替换成指定的字符串。你可以指定一个最多替换次数，否则将替换所有的匹配到的字符串。

第一个参数是正则规则，第二个参数是指定的用来替换的字符串，第三个参数是目标字符串，第四个参数是最多替换次数。

这两个函数的唯一区别是返回值。

sub 返回一个被替换的字符串

sub 返回一个元组，第一个元素是被替换的字符串，第二个元素是一个数字，表明产生了多少次替换。

例，将下面字符串中的’dog’全部替换成’cat’

```
>>> s=’ I have a dog , you have a dog , he have a dog ‘
>>> re.sub( r’dog’ , ‘cat’ , s )
' I have a cat , you have a cat , he have a cat '
```

如果我们只想替换前面两个，则

```
>>> re.sub( r’dog’ , ‘cat’ , s , 2 )
' I have a cat , you have a cat , he have a dog '
```

或者我们想知道发生了多少次替换，则可以使用 subn

```
>>> re.subn( r’dog’ , ‘cat’ , s )
(' I have a cat , you have a cat , he have a cat ', 3)
```

split(rule , target [,maxsplit])

切片函数。使用指定的正则规则在目标字符串中查找匹配的字符串，用它们作为分界，把字符串切片。

第一个参数是正则规则，第二个参数是目标字符串，第三个参数是最多切片次数

返回一个被切完的子字符串的列表

这个函数和 str 对象提供的 split 函数很相似。举个例子，我们想把上例中的字符串被’,’分割开，同时要去掉逗号前后的空格

```
>>> s=’ I have a dog   ,   you have a dog   ,  he have a dog ‘
>>> re.split( ‘\s*,\s*’ , s )
[' I have a dog', 'you have a dog', 'he have a dog ']
```

结果很好。如果使用 str 对象的 split 函数，则由于我们不知道’,’两边会有多少个空格，而不得不对结果再进行一次处理。

escape(string)

这是个功能比较古怪的函数，它的作用是将字符串中的 non-alphanumerics 字符（我已不知道该怎么翻译比较好了）用反义字符的形式显示出来。有时候你可能希望在正则式中匹配一个字符串，不过里面含有很多 re 使用的符号，你要一个一个的修改写法实在有点麻烦，你可以使用这个函数，

例 在目标字符串 s 中匹配’(*+?)’这个子字符串

```
>>> s= ‘111 222 (*+?) 333’
>>> rule= re.escape( r’(*+?)’ )
>>> print rule
\(\*\+\?\)
>>> re.findall( rule , s )
['(*+?)']
```

3 更深入的了解 re 的组与对象

前面对 Python 正则式的组进行了一些简单的介绍，由于还没有介绍到 match 对象，而组又是和 match 对象密切相关的，所以必须将它们结合起来介绍才能充分地说明它们的用途。

不过再详细介绍它们之前，我觉得有必要先介绍一下将规则编译后的生成的 patter 对象

3.1 编译后的 Pattern 对象

将一个正则式，使用 compile 函数编译，不仅是为了提高匹配的速度，同时还能使用一些附加的功能。编译后的结果生成一个 Pattern 对象，这个对象里面有很多函数，他们看起来和 re 模块的函数非常象，它同样有 findall ， match ， search ,finditer ， sub ， subn ， split 这些函数，只不过它们的参数有些小小的不同。一般说来，re 模块函数的第一个参数，即正则规则不再需要了，应为规则就包含在 Pattern 对象中了，编译选项也不再需要了，因为已经被编译过了。因此 re 模块中函数的这两个参数的位置，就被后面的参数取代了。

findall ， match ， search 和 finditer 这几个函数的参数是一样的，除了少了规则和选项两个参数外，它们又加入了另外两个参数，它们是：查找开始位置和查找结束位置，也就是说，现在你可以指定查找的区间，除去你不感兴趣的区间。它们现在的参数形式是：

findall (targetString [, startPos [,endPos]])

finditer (targetString [, startPos [,endPos]])

match (targetString [, startPos [,endPos]])

search (targetString [, startPos [,endPos]])

这些函数的使用和 re 模块的同名函数使用完全一样。所以就不多介绍了。

除了和 re 模块的函数同样的函数外，Pattern 对象还多了些东西，它们是：

flags 查询编译时的选项

pattern 查询编译时的规则

groupindex 规则里的组

这几个不是函数，而是一个值。它们提供你一些规则的信息。比如下面这个例子

```
>>> p=re.compile( r'(?P<word>\b[a-z]+\b)|(?P<num>\b\d+\b)|(?P<id>\b[a-z_]+\w*\b)' , re.I )
>>> p.flags
2
>>> p.pattern
'(?P<word>\\b[a-z]+\\b)|(?P<num>\\b\\d+\\b)|(?P<id>\\b[a-z_]+\\w*\\b)'
>>> p.groupindex
{'num': 2, 'word': 1, 'id': 3}
```

我们来分析一下这个例子：这个正则式是匹配单词、或数字、或一个由字母或’_’开头，后面接字母或数字的一个 ID。我们给这三种情况的规则都包入了一个命名组，分别命名为’word’ , ‘num’ 和 ‘id’。我们规定大小写不敏感，所以使用了编译选项 ‘I’。

编译以后返回的对象为 p，通过 p.flag 我们可以查看编译时的选项，不过它显示的不是’I’，而是一个数值 2 。其实 re.I 是一个整数，2 就是它的值。我们可以查看一下：

```
>>> re.I
2
>>> re.L
4
>>> re.M
8
…
```

每个选项都是一个数值。

通过 p.pattern 可以查看被编译的规则是什么。使用 print 的话会更好看一些

```
>>> print p.pattern
(?P<word>\b[a-z]+\b)|(?P<num>\b\d+\b)|(?P<id>\b[a-z_]+\w*\b)
```

看，和我们输入的一样。

接下来的 p.groupindex 则是一个字典，它包含了规则中的所有命名组。字典的 key 是名字，values 是组的序号。由于字典是以名字作为 key，所以一个无命名的组不会出现在这里。

3.2 组与 Match 对象

组与 Match 对象是 Python 正则式的重点。只有掌握了组和 Match 对象的使用，才算是真正学会了 Python 正则式。

3.2.1 组的名字与序号

正则式中的每个组都有一个序号，它是按定义时从左到右的顺序从 1 开始编号的。其实，re 的正则式还有一个 0 号组，它就是整个正则式本身。

我们来看个例子

```
>>> p=re.compile( r'(?P<name>[a-z]+)\s+(?P<age>\d+)\s+(?P<tel>\d+).*' , re.I )
>>> p.groupindex
{'age': 2, 'tel': 3, 'name': 1}
>>> s='Tom 24 88888888  <='
>>> m=p.search(s)
>>> m.groups()            # 看看匹配的各组的情况
('Tom', '24', '8888888')
>>> m.group('name')       # 使用组名获取匹配的字符串
```

‘Tom’

```
>>> m.group( 1 )            # 使用组序号获取匹配的字符串，同使用组名的效果一样
>>> m.group(0)              # 0 组里面是什么呢？
'Tom 24 88888888  <='
```

原来 0 组就是整个正则式,包括没有被包围到组里面的内容。当获取 0 组的时候，你可以不写这个参数。m.group(0)和 m.group()的效果是一样的：

```
>>> m.group()
'Tom 24 88888888  <='
```

接下来看看更多的 Match 对象的方法，看看我们能做些什么。

3.2.2 Match 对象的方法

group([index|id]) 获取匹配的组，缺省返回组 0,也就是全部值

groups() 返回全部的组

groupdict() 返回以组名为 key，匹配的内容为 values 的字典

接上例：

```
>>> m.groupindex()
{'age': '24', 'tel': '88888888', 'name': 'Tom'}
```

start([group])获取匹配的组的开始位置

end([group]) 获取匹配的组的结束位置

span([group]) 获取匹配的组的（开始，结束）位置

expand(template) 根据一个模版用找到的内容替换模版里的相应位置

这个功能比较有趣，它根据一个模版来用匹配到的内容替换模版中的相应位置，组成一个新的字符串返回。它使用\g<index|name>或 \index 来指示一个组。

接上例

```
>>> m.expand(r'name is \g<1> , age is \g<age> , tel is \3')
'name is Tom , age is 24 , tel is 88888888'
```

除了以上这些函数外，Match 对象还有些属性

pos 搜索开始的位置参数

endpos 搜索结束的位置参数

这两个是使用 findall 或 match 等函数时，传入的参数。在上面这个例子里，我们没有指定开始和结束位置，那么缺省的开始位置就是 0,结束位置就是最后。

```
>>> m.pos
0
>>> m.endpos
19
```

lastindex 最后匹配的组的序号

```
>>> m.lastindex
3
```

lastgroup 最后匹配的组名

```
>>> m.lastgroup
'tel'
```

re 产生这个匹配的 Pattern 对象，可以认为是个逆引用

```
>>> m.re.pattern
'(?P<name>[a-z]+)\\s+(?P<age>\\d+)\\s+(?P<tel>\\d+).*'
```

得到了产生这个匹配的规则

string 匹配的目标字符串

```
>>> m.string
'Tom 24 88888888  <='
```

正则表达式语法

正则表达式（RE）指定一个与之匹配的字符集合；本模块所提供的函数，将可以用来检查所给的字符串是否与指定的正则表达式匹配。
正则表达式可以被连接，从而形成新的正则表达式；例如 A 和 B 都是正则表达式，那么 AB 也是正则表达式。一般地，如果字符串 *p* 与 A 匹配， *q* 与 B 匹配的话，那么字符串 *pq* 也会与 AB 匹配，但 A 或者 B 里含有边界限定条件或者命名组操作的情况除外。也就是说，复杂的正则表达式可以用简单的连接而成。
正则表达式可以包含特殊字符和普通字符，大部分字符比如'A'，'a'和'0'都是普通字符，如果做为正则表达式，它们将匹配它们本身。由于正则表达式可以连接，所以连接多个普通字符而成的正则表达式 last 也将匹配'last'。（后面将用不带引号的表示正则表达式，带引号的表示字符串）
下面就来介绍正则表达式的特殊字符：
'.'
点号，在普通模式，它匹配除换行符外的任意一个字符；如果指定了 **DOTALL** 标记，匹配包括换行符以内的任意一个字符。
'^'
尖尖号，匹配一个字符串的开始，在 **MULTILINE** 模式下，也将匹配任意一个新行的开始。
'$'
美元符号，匹配一个字符串的结尾或者字符串最后面的换行符，在 **MULTILINE** 模式下，也匹配任意一行的行尾。也就是说，普通模式下，foo.$去搜索'foo1\nfoo2\n'只会找到’foo2’，但是在 **MULTILINE** 模式，还能找到 ‘foo1’，而且就用一个 $ 去搜索'foo\n'的话，会找到两个空的匹配：一个是最后的换行符，一个是字符串的结尾，演示：
```
&gt;&gt;&gt;re.findall('(foo.$)',
'foo1\nfoo2\n')['foo2']&gt;&gt;&gt;re.findall('(foo.$)', 'foo1\nfoo2\n',
re.MULTILINE)['foo1', 'foo2']&gt;&gt;&gt;re.findall('($)', 'foo\n')['',
'']
```
'*'
星号，指定将前面的 RE 重复 0 次或者任意多次，而且总是试图尽量多次地匹配。
'+'
加号，指定将前面的 RE 重复 1 次或者任意多次，而且总是试图尽量多次地匹配。
'?'
问号，指定将前面的 RE 重复 0 次或者 1 次，如果有的话，也尽量匹配 1 次。
*?， +?， ??
从前面的描述可以看到'*'，'+'和'?'都是*贪婪的*，但这也许并不是我们说要的，所以，可以在后面加个问号，将策略改为 *非贪婪*，只匹配尽量少的 RE。示例，体会两者的区别：
```
&gt;&gt;&gt;re.findall('', '<H1>title</H1>')['H1&gt;title</H1>&gt;&gt;
re.findall('', '<H1>title</H1>') ['H1', '/H1']
```
{m}
m 是一个数字，指定将前面的 RE 重复 m 次。

{m,n}

m 和 n 都是数字，指定将前面的 RE 重复 m 到 n 次，例如 a{3,5}匹配 3 到 5 个连续的 a。注意，如果省略 m，将匹配 0 到 n 个前面的 RE；如果省略 n，将匹配 n 到无穷多个前面的 RE；当然中间的逗号是不能省略的，不然就变成前面那种形式了。

{m,n}?

前面说的{m,n}，也是贪婪的，a{3,5}如果有 5 个以上连续 a 的话，会匹配 5 个，这个也可以通过加问号改变。 a{3,5}?如果可能的话，将只匹配 3 个 a。

'\'

反斜杆，转义'*'，'?'等特殊字符，或者指定一个特殊序列（下面会详述）

由于之前所述的原因，强烈建议用 raw 字符串来表述正则。

[]

方括号，用于指定一个字符的集合。可以单独列出字符，也可以用'-'连接起止字符以表示一个范围。特殊字符在中括号里将失效，比如 [akm$]就表示字符'a','k','m',或 '$'，在这里$也变身为普通字符了。[a-z]匹配任意一个小写字母，[a-zA-Z0-9]匹配任意一个字母或数字。如果你要匹配 ']'或'-'本身，你需要加反斜杆转义，或者是将其置于中括号的最前面，比如[]]可以匹配 ']'

你还可以对一个字符集合*取反*，以匹配任意不在这个字符集合里的字符，取反操作用一个'^'放在集合的最前面表示，放在其他地方的'^'将不会起特殊作用。例如[^5]将匹配任意不是 '5'的字符；[^^]将匹配任意不是'^'的字符。

注意：在中括号里，+、*、(、)这类字符将会失去特殊含义，仅作为普通字符。反向引用也不能在中括号内使用。

'|'

管道符号，A 和 B 是任意的 RE，那么 A|B 就是匹配 A 或者 B 的一个新的 RE。任意个数的 RE 都可以像这样用管道符号间隔连接起来。这种形式可以被用于 **组**中（后面将详述）。对于目标字符串，被'|'分割的 RE 将自左至右一一被测试，一旦有一个测试成功，后面的将不再被测试，即使后面的 RE 可能可以匹配更长的串，换句话说， '|'操作符是非贪婪的。要匹配字面意义上的'|'，可以用反斜杆转义：\|，或是包含在反括号内： [|]。

(...)

匹配圆括号里的 RE 匹配的内容，并指定**组**的开始和结束位置。组里面的内容可以被提取，也可以采用 \number 这样的特殊序列，被用于后续的匹配。要匹配字面意义上的'('和')'，可以用反斜杆转义： \(、\)，或是包含在反括号内：[(]、[)]。

(?...)

这是一个表达式的扩展符号。'?'后的第一个字母决定了整个表达式的语法和含义，除了(?P...)以外，表达式不会产生一个新的组。下面介绍几个目前已被支持的扩展：

(?iLmsux)

'i'、'L'、'm'、's'、 'u'、'x'里的一个或多个字母。表达式不匹配任何字符，但是指定相应的标志：**re.I**(忽略大小写)、 **re.L**(依赖 locale)、**re.M**(多行模式)、**re.S**(.匹配所有字符)、 **re.U**(依赖 Unicode)、**re.X**(详细模式)。关于各个模式的区别，下面会

有专门的一节来介绍的。使用这个语法可以代替在 re.compile()的时候或者调用的时候指定 *flag* 参数。

例如，上面举过的例子，可以改写成这样（和指定了 re.MULTILINE 是一样的效果）：

```
&gt;&gt;&gt;re.findall('(?m)(foo.$)', 'foo1\nfoo2\n')['foo1', 'foo2']
```

另外，还要注意(?x)标志如果有的话，要放在最前面。

(?:...)

匹配内部的 RE 所匹配的内容，但是不建立组。

(?P<name>...)

和普通的圆括号类似，但是子串匹配到的内容将可以用命名的 *name* 参数来提取。组的 *name* 必须是有效的 python 标识符，而且在本表达式内不重名。命名了的组和普通组一样，也用数字来提取，也就是说名字只是个额外的属性。

演示一下：

```
&gt;&gt;&gt; m=re.match('(?P<var>[a-zA-Z_]\w*)', 'abc=123')&gt;&gt;&gt;
m.group('var')'abc'&gt;&gt;&gt; m.group(1)'abc'
```

(?P=name)

匹配之前以 *name* 命名的组里的内容。

演示一下：

```
&gt;&gt;&gt;re.match('&lt;(?P\w*)&gt;.*', '<h1>xxx</h2>')#这个不匹配
&gt;&gt;&gt;re.match('&lt;(?P\w*)&gt;.*', '<h1>xxx</h1>')#这个匹配
```

(?#...)

注释，圆括号里的内容会被忽略。

(?=...)

如果 ... 匹配接下来的字符，才算匹配，但是并不会消耗任何被匹配的字符。例如 Isaac (?=Asimov) 只会匹配后面跟着 'Asimov' 的 'Isaac '，这个叫做“前瞻断言”。

(?!...)

和上面的相反，只匹配接下来的字符串不匹配 ... 的串，这叫做“反前瞻断言”。

(?<=...)

只有当当前位置之前的字符串匹配 ... ，整个匹配才有效，这叫“后顾断言”。

(?<=abc)def 会找到 'abcdef'，因为会后向查找 3 个字符，看是否为 abc。所以内置的子 RE，需要是固定长度的，比如可以是 abc、a|b，但不能是 a*、a{3,4}。注意这种 RE 永远不会匹配到字符串的开头。举个例子，找到连字符（ '-'）后的单词：

```
&gt;&gt;&gt; m = re.search('(?&gt;&gt; m.group(0) 'egg'
```

(?<!...)

同理，这个叫做“反后顾断言”，子 RE 需要固定长度的，含义是前面的字符串不匹配 ... 整个才算匹配。

(?(id/name)yes-pattern|no-pattern)

如有由 *id* 或者 *name* 指定的组存在的话，将会匹配 yes-pattern，否则将会匹配 no-pattern，通常情况下 no-pattern 也可以省略。例如：()可以匹

配 '<user@host.com>' 和 'user@host.com'，但是不会匹配 '<user@host.com'。

下面列出以'\'开头的特殊序列。如果某个字符没有在下面列出，那么 RE 的结果会只匹配那个字母本身，比如， \$只匹配字面意义上的'$'。

\number

匹配 number 所指的组相同的字符串。组的序号从 1 开始。例如：(.+) \1 可以匹配'the the'和'55 55'，但不匹配'the end'。这种序列在一个正则表达式里最多可以有 99 个，如果 *number* 以 0 开头，或是有 3 位以上的数字，就会被当做八进制表示的字符了。同时，这个也不能用于方括号内。

\A

只匹配字符串的开始。

\b

匹配单词边界（包括开始和结束），这里的“单词”，是指连续的字母、数字和下划线组成的字符串。注意，\b 的定义是 \w 和\W 的交界，所以精确的定义有赖于 UNICODE 和 LOCALE 这两个标志位。

\B

和\b 相反，\B 匹配非单词边界。也依赖于 UNICODE 和 LOCALE 这两个标志位。

\d

未指定 UNICODE 标志时，匹配数字，等效于：[0-9]。指定了 UNICODE 标志时，还会匹配其他 Unicode 库里描述为字符串的符号。便于理解，举个例子（好不容易找的例子啊，呵呵）：

```
#\u2076\和 u2084 分别是上标的 6 和下标的 4，属于 unicode 的 DIGIT&gt;&gt;&gt;
unistr = u'\u2076\u2084abc'&gt;&gt;&gt;print unistr ⁶₄abc
&gt;&gt;&gt;printre.findall('\d+', unistr, re.U)[0] ⁶₄
```

\D

和\d 相反，不多说了。

\s

当未指定 UNICODE 和 LOCALE 这两个标志位时，匹配任何空白字符，等效于 [\t\n\r\f\v]。如果指定了 LOCALE，则还要加 LOCALE 相关的空白字符；如果指定了 UNICODE，还要加上 UNICODE 空白字符，如较常见的空宽度连接空格（\uFEFF）、零宽度非连接空格(\u200B)等。

\S

和\s 相反，也不多说。

\w

当未指定 UNICODE 和 LOCALE 这两个标志位时，等效于[a-zA-Z0-9_]。当指定了 LOCALE 时，为[0-9_]加上当前 LOCAL 指定的字母。当指定了 UNICODE 时，为 [0-9_] 加上 UNICODE 库里的所有字母。

\W

和\w 相反，不多说。

\Z

只匹配字符串的结尾。

匹配之于搜索

python 提供了两种基于正则表达式的操作：匹配（match）从字符串的开始检查字符串是否个正则匹配。而搜索（search）检查字符串任意位置是否有匹配的子串（perl 默认就是如此）。

注意，即使 search 的正则以'^'开头，match 和 search 也还是有许多不同的。

```
&gt;&gt;&gt;re.match("c", "abcdef")# 不匹配&gt;&gt;&gt;re.search("c", "abcdef")# 匹配
```

模块的属性和方法

re.compile(*pattern*[, *flags*])

把一个正则表达式 *pattern* 编译成正则对象，以便可以用正则对象的 **match** 和 **search** 方法。

得到的正则对象的行为（也就是模式）可以用 *flags* 来指定，值可以由几个下面的值 OR 得到。

以下两段内容在语法上是等效的：

```
prog = re.compile(pattern) result = prog.match(string)
result = re.match(pattern, string)
```

区别是，用了 **re.compile** 以后，正则对象会得到保留，这样在需要多次运用这个正则对象的时候，效率会有较大的提升。再用上面用过的例子来演示一下，用相同的正则匹配相同的字符串，执行 100 万次，就体现出 compile 的效率了（数据来自我那 1.86G CPU 的神舟本本）：

```
&gt;&gt;&gt;timeit.timeit( ... setup='''import re; reg =
re.compile('&lt;(?P\w*)&gt;.*')''', ...
stmt='''reg.match('<h1>xxx</h1>')''', ...
number=1000000)1.2062149047851562&gt;&gt;&gt;timeit.timeit( ...
setup='''import re''', ... stmt='''re.match('&lt;(?P\w*)&gt;.*',
'<h1>xxx</h1>')''', ... number=1000000)4.4380838871002197
```

re.I

re.IGNORECASE

让正则表达式忽略大小写，这样一来，[A-Z]也可以匹配小写字母了。此特性和 locale 无关。

re.L

re.LOCALE

让\w、\W、\b、\B、 \s 和\S 依赖当前的 locale。

re.M

re.MULTILINE

影响'^'和'$'的行为，指定了以后，'^'会增加匹配每行的开始（也就是换行符后的位置）；'$'会增加匹配每行的结束（也就是换行符前的位置）。

re.S

re.DOTALL

影响'.'的行为，平时'.'匹配除换行符以外的所有字符，指定了本标志以后，也可以匹配换行符。

re.U

re.UNICODE

让\w、\W、\b、\B、 \d、\D、\s 和\S 依赖 Unicode 库。

re.X

re.VERBOSE

运用这个标志，你可以写出可读性更好的正则表达式：除了在方括号内的和被反斜杠转义的以外的所有空白字符，都将被忽略，而且每行中，一个正常的井号后的所有字符也被忽略，这样就可以方便地在正则表达式内部写注释了。也就是说，下面两个正则表达式是等效的：

```
a = re.compile(r"""\d + # the integral part \. # the decimal point \d * # some fractional digits""", re.X) b = re.compile(r"\d+\.\d*")
```

re.search(*pattern*, *string*[, *flags*])

扫描 *string*，看是否有个位置可以匹配正则表达式 *pattern*。如果找到了，就返回一个 **MatchObject** 的实例，否则返回 **None**，注意这和找到长度为 0 的子串含义是不一样的。搜索过程受 *flags* 的影响。

re.match(*pattern*, *string*[, *flags*])

如果字符串 *string* 的开头和正则表达式 *pattern* 匹配的话，返回一个相应的 **MatchObject** 的实例，否则返回 **None**

注意：要在字符串的任意位置搜索的话，需要使用上面的 **search()**。

re.split(*pattern*, *string*[, *maxsplit=0*])

用匹配 *pattern* 的子串来分割 *string*，如果 *pattern* 里使用了圆括号，那么被 *pattern* 匹配到的串也将作为返回值列表的一部分。如果 *maxsplit* 不为 0，则最多被分割为 *maxsplit* 个子串，剩余部分将整个地被返回。

```
&gt;&gt;&gt;re.split('\W+', 'Words, words, words.')['Words', 'words', 'words', '']&gt;&gt;&gt;re.split('(\W+)', 'Words, words, words.')['Words', ', ', 'words', ', ', 'words', '.', '']&gt;&gt;&gt;re.split('\W+', 'Words, words, words.', 1)['Words', 'words, words.']
```

如果正则有圆括号，并且可以匹配到字符串的开始位置的时候，返回值的第一项，会多出一个空字符串。匹配到字符结尾也是同样的道理：

```
&gt;&gt;&gt;re.split('(\W+)', '...words, words...')['', '...', 'words', ', ', 'words', '...', '']
```

注意，*split* 不会被零长度的正则所分割，例如：

```
&gt;&gt;&gt;re.split('x*', 'foo')['foo']&gt;&gt;&gt;re.split("(?m)^$", "foo\n\nbar\n")['foo\n\nbar\n']
```

re.findall(*pattern*, *string*[, *flags*])

以列表的形式返回 *string* 里匹配 *pattern* 的不重叠的子串。*string* 会被从左到右依次扫描，返回的列表也是从左到右一次匹配到的。如果 *pattern* 里含有**组**的话，那么会返回匹配到的组的列表；如果 *pattern* 里有多个组，那么各组会先组成一个元组，然后返回值将是一个元组的列表。

由于这个函数不会涉及到 **MatchObject** 之类的概念，所以，对新手来说，应该是最好理解也最容易使用的一个函数了。下面就此来举几个简单的例子：

```
#简单的 findall&gt;&gt;&gt;re.findall('\w+', 'hello, world!')['hello', 'world']#这个返回的就是元组的列表
&gt;&gt;&gt;re.findall('(\d+)\.(\d+)\.(\d+)\.(\d+)', 'My IP is 192.168.0.2, and your is 192.168.0.3.')[('192', '168', '0', '2'), ('192', '168', '0', '3')]
```

re. finditer(*pattern*, *string*[, *flags*])

和上面的 **findall()**类似，但返回的是 **MatchObject** 的实例的迭代器。

还是例子说明问题：

```
>>> for m in re.finditer('\w+', 'hello, world!'):
… print m.group()
…
hello
world
```

re.sub(*pattern*, *repl*, *string*[, *count*])

替换，将 *string* 里，匹配 *pattern* 的部分，用 *repl* 替换掉，最多替换 *count* 次（剩余的匹配将不做处理），然后返回替换后的字符串。如果 *string* 里没有可以匹配 *pattern* 的串，将被原封不动地返回。 *repl* 可以是一个字符串，也可以是一个函数（也可以参考我以前的例子）。如果 *repl* 是个字符串，则其中的反斜杆会被处理过，比如 \n 会被转成换行符，反斜杆加数字会被替换成相应的组，比如 \6 表示 *pattern* 匹配到的第 6 个组的内容。

例子：

```
&gt;&gt;&gt;re.sub(r'def\s+([a-zA-Z_][a-zA-Z_0-9]*)\s*\(\s*\):', ... r'static PyObject*\npy_\1(void)\n{', ... 'def myfunc():')'static PyObject*\npy_myfunc(void)\n{'
```

如果 *repl* 是个函数，每次 *pattern* 被匹配到的时候，都会被调用一次，传入一个匹配到的 **MatchObject** 对象，需要返回一个字符串，在匹配到的位置，就填入返回的字符串。

例子：

```
&gt;&gt;&gt;def dashrepl(matchobj): ... if matchobj.group(0) == '-':
return' ' ... else: return'-'&gt;&gt;&gt;re.sub('-{1,2}', dashrepl,
'pro----gram-files')'pro--gram files'
```

零长度的匹配也会被替换，比如：

```
&gt;&gt;&gt;re.sub('x*', '-', 'abcxxd')'-a-b-c-d-'
```

特殊地，在替换字符串里，如果有\g 这样的写法，将匹配正则的命名组（前面介绍过的，(?P...)这样定义出来的东西）。 \g 这样的写法，也是数字的组，也就是说，\g 一般和 \2 是等效的，但是万一你要在 \2 后面紧接着写上字面意义的 0，你就不能写成\20 了（因为这代表第 20 个组），这时候必须写成 \g0，另外，\g 代表匹配到的整个子串。

例子：

```
&gt;&gt;&gt;re.sub('-(\d+)-', '-\g0\g',
'a-11-b-22-c')'a-110-11-b-220-22-c'
```

re.subn(*pattern*, *repl*, *string*[, *count*])

跟上面的 sub()函数一样，只是它返回的是一个元组 (新字符串, 匹配到的次数)，还是用例子说话：

```
&gt;&gt;&gt;re.subn('-(\d+)-', '-\g0\g',
'a-11-b-22-c')('a-110-11-b-220-22-c', 2)
```

re.escape(*string*)

把 *string* 中，除了字母和数字以外的字符，都加上反斜杆。

```
&gt;&gt;&gt;printre.escape('abc123_@#$') abc123\_\@\#\$
```

exception re.error

如果字符串不能被成功编译成正则表达式或者正则表达式在匹配过程中出错了，都会抛出此异常。但是如果正则表达式没有匹配到任何文本，是不会抛出这个异常的。

正则对象

正则对象由 re.compile()返回。它有如下的属性和方法。

match(*string*[, *pos*[, *endpos*]])

作用和模块的 **match()**函数类似，区别就是后面两个参数。

pos 是开始搜索的位置，默认为 0。*endpos* 是搜索的结束位置，如果 *endpos* 比 *pos* 还小的话，结果肯定是空的。也就是说只有 *pos* 到 *endpos*-1 位置的字符串将会被搜索。

例子：

```
&gt;&gt;&gt; pattern = re.compile("o")&gt;&gt;&gt; pattern.match("dog")#
开始位置不是 o，所以不匹配&gt;&gt;&gt; pattern.match("dog", 1)# 第二个字符是
o，所以匹配
```

search(*string*[, *pos*[, *endpos*]])

作用和模块的 **search()**函数类似，*pos* 和 *endpos* 参数和上面的 **match()**函数类似。

split(*string*[, *maxsplit*=0])

findall(*string*[, *pos*[, *endpos*]])

finditer(*string*[, *pos*[, *endpos*]])

sub(*repl*, *string*[, *count=0*])

subn(*repl*, *string*[, *count=0*])

这几个函数，都和模块的相应函数一致。

flags

编译本 RE 时，指定的标志位，如果未指定任何标志位，则为 0。

```
&gt;&gt;&gt; pattern = re.compile("o", re.S|re.U)&gt;&gt;&gt;
pattern.flags48
```

groups

RE 所含有的组的个数。

groupindex

一个字典，定义了命名组的名字和序号之间的关系。

例子：

这个正则有 3 个组，如果匹配到，第一个叫区号，最后一个叫分机号，中间的那个未命名

```
&gt;&gt;&gt; pattern = re.compile("(?P\d+)-(\d+)-(?P\d+)")&gt;&gt;&gt;
pattern.groups3&gt;&gt;&gt; pattern.groupindex{'fenjihao': 3, 'quhao': 1}
```

pattern

建立本 RE 的原始字符串，相当于源代码了，呵呵。

还是上面这个正则，可以看到，会原样返回：

```
&gt;&gt;&gt;print pattern.pattern(?P\d+)-(\d+)-(?P\d+)
```

Match 对象

re.MatchObject 被用于布尔判断的时候，始终返回 True，所以你用 **if** 语句来判断某个 **match()** 是否成功是安全的。

它有以下方法和属性：

expand(*template*)

用 *template* 做为模板，将 **MatchObject** 展开，就像 **sub()**里的行为一样，看例子：

```
&gt;&gt;&gt; m = re.match('a=(\d+)', 'a=100')&gt;&gt;&gt; m.expand('above
a is \g')'above a is 100'&gt;&gt;&gt; m.expand(r'above a is \1')'above a
is 100'
```

group([*group1*, ...])

返回一个或多个子组。如果参数为一个，就返回一个子串；如果参数有多个，就返回多个子串注册的元组。如果不传任何参数，效果和传入一个 0 一样，将返回整个匹配。如果某个 *groupN* 未匹配到，相应位置会返回 **None**。如果某个 *groupN* 是负数或者大于 group 的总数，则会抛出 **IndexError** 异常。

```
&gt;&gt;&gt; m = re.match(r"(\w+) (\w+)", "Isaac Newton,
physicist")&gt;&gt;&gt; m.group(0)# 整个匹配'Isaac Newton'&gt;&gt;&gt;
m.group(1)# 第一个子串'Isaac'&gt;&gt;&gt; m.group(2)# 第二个子串
'Newton'&gt;&gt;&gt; m.group(1, 2)# 多个子串组成的元组('Isaac', 'Newton')
```

如果有其中有用(?P...)这种语法命名过的子串的话，相应的 *groupN* 也可以是名字字符串。例如：

```
&gt;&gt;&gt; m = re.match(r"(?P\w+) (?P\w+)", "Malcolm Reynolds")&gt;&gt;&gt; m.group('first_name')'Malcolm'&gt;&gt;&gt; m.group('last_name')'Reynolds'
```

如果某个组被匹配到多次，那么只有最后一次的数据，可以被提取到：

```
&gt;&gt;&gt; m = re.match(r"(..)+", "a1b2c3")# 匹配到 3 次&gt;&gt;&gt; m.group(1)# 返回的是最后一次'c3'
```

groups([*default*])

返回一个由所有匹配到的子串组成的元组。*default* 参数，用于给那些没有匹配到的组做默认值，它的默认值是 **None**

例如：

```
&gt;&gt;&gt; m = re.match(r"(\d+)\.(\d+)", "24.1632")&gt;&gt;&gt; m.groups()('24', '1632')
```

default 的作用：

```
&gt;&gt;&gt; m = re.match(r"(\d+)\.?(\d+)?", "24")&gt;&gt;&gt; m.groups()# 第二个默认是 None('24', None)&gt;&gt;&gt; m.groups('0')# 现在默认是 0 了('24', '0')
```

groupdict([*default*])

返回一个包含所有命名组的名字和子串的字典，*default* 参数，用于给那些没有匹配到的组做默认值，它的默认值是 **None**，例如：

```
&gt;&gt;&gt; m = re.match(r"(?P\w+) (?P\w+)", "Malcolm Reynolds")&gt;&gt;&gt; m.groupdict(){'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

start([**group**])

end([*group*])

返回的是：被组 *group* 匹配到的子串在原字符串中的位置。如果不指定 *group* 或 *group* 指定为 0，则代表整个匹配。如果 *group* 未匹配到，则返回 -1。

对于指定的 m 和 g，m.group(g)和 m.string[m.start(g):m.end(g)]等效。

注意：如果 *group* 匹配到空字符串，*m.start(group)*和 *m.end(group)*将相等。

例如：

```
&gt;&gt;&gt; m = re.search('b(c?)', 'cba')&gt;&gt;&gt; m.start(0)1&gt;&gt;&gt; m.end(0)2&gt;&gt;&gt; m.start(1)2&gt;&gt;&gt; m.end(1)2
```

下面是一个把 email 地址里的“remove_this”去掉的例子：

```
&gt;&gt;&gt;email = "tony@tiremove_thisger.net"&gt;&gt;&gt; m = re.search("remove_this", email)&gt;&gt;&gt;email[:m.start()] + email[m.end():]'tony@tiger.net'
```

span([*group*])
返回一个元组： (m.start(group), m.end(group))
pos
就是传给 RE 对象的 **search()**或 **match()**方法的参数 *pos*，代表 RE 开始搜索字符串的位置。
endpos
就是传给 RE 对象的 **search()**或 **match()**方法的参数 *endpos*，代表 RE 搜索字符串的结束位置。
lastindex
最后一次匹配到的组的数字序号，如果没有匹配到，将得到 **None**。
例如：(a)b、((a)(b))和((ab))正则去匹配 'ab'的话，得到的 **lastindex** 为 1。而用(a)(b)去匹配 'ab'的话，得到的 **lastindex** 为 2。
lastgroup
最后一次匹配到的组的名字，如果没有匹配到或者最后的组没有名字，将得到 **None**。
re
得到本 Match 对象的正则表达式对象，也就是执行 **search()**或 **match()**的对象。
string
传给 **search()**或 **match()**的字符串。
后面的例子就略了吧，文中已经加了很多我自己的例子了，需要更多例子的话，参照英文原文吧。
最后，感谢我的老婆辛苦地帮我校对，哈哈。
Python 中的 random 模块用于生成随机数。下面介绍一下 random 模块中最常用的几个函数。

random.random
random.random()用于生成一个 0 到 1 的随机符点数: 0 <= n < 1.0

random.uniform
random.uniform 的函数原型为：random.uniform(a, b)，用于生成一个指定范围内的随机符点数，两个参数其中一个是上限，一个是下限。如果 a > b，则生成的随机数 n: b <= n <= a。如果 a <b， 则 a <= n <= b。

```
print random.uniform(10, 20)
print random.uniform(20, 10)
#---- 结果（不同机器上的结果不一样）
#18.7356606526
#12.5798298022
print random.uniform(10, 20)
print random.uniform(20, 10)
#---- 结果（不同机器上的结果不一样）
#18.7356606526
```

```
#12.5798298022
```

random.randint

random.randint()的函数原型为：random.randint(a, b)，用于生成一个指定范围内的整数。其中参数 a 是下限，参数 b 是上限，生成的随机数 n: a <= n <= b

```
print random.randint(12, 20)  #生成的随机数 n: 12 <= n <= 20
print random.randint(20, 20)  #结果永远是 20
#print random.randint(20, 10)  #该语句是错误的。下限必须小于上限。
print random.randint(12, 20)  #生成的随机数 n: 12 <= n <= 20
print random.randint(20, 20)  #结果永远是 20
#print random.randint(20, 10)  #该语句是错误的。下限必须小于上限。
```

random.randrange

random.randrange 的函数原型为：random.randrange([start], stop[, step])，从指定范围内，按指定基数递增的集合中 获取一个随机数。如：random.randrange(10, 100, 2)，结果相当于从[10, 12, 14, 16, ... 96, 98]序列中获取一个随机数。random.randrange(10, 100, 2)在结果上与 random.choice(range(10, 100, 2) 等效。

random.choice

random.choice 从序列中获取一个随机元素。其函数原型为：random.choice(sequence)。参数 sequence 表示一个有序类型。这里要说明 一下：sequence 在 python 不是一种特定的类型，而是泛指一系列的类型。list, tuple, 字符串都属于 sequence。有关 sequence 可以查看 python 手册数据模型这一章，也可以参考：http://www.17xie.com/read-37422.html 。下面是使用 choice 的一些例子：

```
print random.choice("学习 Python")
print random.choice(["JGood", "is", "a", "handsome", "boy"])
print random.choice(("Tuple", "List", "Dict"))
print random.choice("学习 Python")
print random.choice(["JGood", "is", "a", "handsome", "boy"])
print random.choice(("Tuple", "List", "Dict"))
```

random.shuffle

random.shuffle 的函数原型为：random.shuffle(x[, random])，用于将一个列表中的元素打乱。如:p = ["Python", "is", "powerful", "simple", "and so on..."]

```
random.shuffle(p)
print p
#---- 结果（不同机器上的结果可能不一样。）
#['powerful', 'simple', 'is', 'Python', 'and so on...']
p = ["Python", "is", "powerful", "simple", "and so on..."]
random.shuffle(p)
print p
```

```
#---- 结果（不同机器上的结果可能不一样。）
#['powerful', 'simple', 'is', 'Python', 'and so on...']
```

random.sample

random.sample 的函数原型为：random.sample(sequence, k)，从指定序列中随机获取指定长度的片断。sample 函数不会修改原有序列。

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
slice = random.sample(list, 5)  #从 list 中随机获取 5 个元素，作为一个片断返回
print slice
print list #原有序列并没有改变。
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
slice = random.sample(list, 5)  #从 list 中随机获取 5 个元素，作为一个片断返回
print slice
print list #原有序列并没有改变。
```

上面这些方法是 random 模块中最常用的，在 Python 手册中，还介绍其他的方法。感兴趣的朋友可以通过查询 Python 手册了解更详细的信息。

Python 中的 random 模块用于生成随机数。下面介绍一下 random 模块中最常用的几个函数。

random.random

random.random()用于生成一个 0 到 1 的随机符点数: 0 <= n < 1.0

random.uniform

random.uniform 的函数原型为：random.uniform(a, b)，用于生成一个指定范围内的随机符点数，两个参数其中一个是上限，一个是下限。如果 a > b，则生成的随机数 n: b <= n <= a。如果 a <b， 则 a <= n <= b。

```
print random.uniform(10, 20)
print random.uniform(20, 10)
#---- 结果（不同机器上的结果不一样）
#18.7356606526
#12.5798298022
print random.uniform(10, 20)
print random.uniform(20, 10)
#---- 结果（不同机器上的结果不一样）
#18.7356606526
#12.5798298022
```

random.randint

random.randint()的函数原型为：random.randint(a, b)，用于生成一个指定范围内的整数。其中参数 a 是下限，参数 b 是上限，生成的随机数 n: a <= n <= b

```
print random.randint(12, 20)  #生成的随机数 n: 12 <= n <= 20
print random.randint(20, 20)  #结果永远是 20
#print random.randint(20, 10)  #该语句是错误的。下限必须小于上限。
print random.randint(12, 20)  #生成的随机数 n: 12 <= n <= 20
print random.randint(20, 20)  #结果永远是 20
#print random.randint(20, 10)  #该语句是错误的。下限必须小于上限。
```

random.randrange

random.randrange 的函数原型为：random.randrange([start], stop[, step])，从指定范围内，按指定基数递增的集合中 获取一个随机数。如：random.randrange(10, 100, 2)，结果相当于从[10, 12, 14, 16, ... 96, 98]序列中获取一个随机数。random.randrange(10, 100, 2)在结果上与 random.choice(range(10, 100, 2) 等效。

random.choice

random.choice 从序列中获取一个随机元素。其函数原型为：random.choice(sequence)。参数 sequence 表示一个有序类型。这里要说明 一下：sequence 在 python 不是一种特定的类型，而是泛指一系列的类型。list, tuple, 字符串都属于 sequence。有关 sequence 可以查看 python 手册数据模型这一章，也可以参考：http://www.17xie.com/read-37422.html 。下面是使用 choice 的一些例子：

```
print random.choice("学习 Python")
print random.choice(["JGood", "is", "a", "handsome", "boy"])
print random.choice(("Tuple", "List", "Dict"))
print random.choice("学习 Python")
print random.choice(["JGood", "is", "a", "handsome", "boy"])
print random.choice(("Tuple", "List", "Dict"))
```

random.shuffle

random.shuffle 的函数原型为：random.shuffle(x[, random])，用于将一个列表中的元素打乱。如:p = ["Python", "is", "powerful", "simple", "and so on..."]

```
random.shuffle(p)
print p
#---- 结果（不同机器上的结果可能不一样。）
#['powerful', 'simple', 'is', 'Python', 'and so on...']
p = ["Python", "is", "powerful", "simple", "and so on..."]
random.shuffle(p)
print p
#---- 结果（不同机器上的结果可能不一样。）
#['powerful', 'simple', 'is', 'Python', 'and so on...']
```

random.sample

random.sample 的函数原型为：random.sample(sequence, k)，从指定序列中随机

获取指定长度的片断。sample 函数不会修改原有序列。

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
slice = random.sample(list, 5)  #从 list 中随机获取 5 个元素，作为一个片断返回
print slice
print list #原有序列并没有改变。
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
slice = random.sample(list, 5)  #从 list 中随机获取 5 个元素，作为一个片断返回
print slice
print list #原有序列并没有改变。
```

上面这些方法是 random 模块中最常用的，在 Python 手册中，还介绍其他的方法。感兴趣的朋友可以通过查询 Python 手册了解更详细的信息。

Python 中的 random 模块用于生成随机数。下面介绍一下 random 模块中最常用的几个函数。

random.random

random.random()用于生成一个 0 到 1 的随机符点数: 0 <= n < 1.0

random.uniform

random.uniform 的函数原型为：random.uniform(a, b)，用于生成一个指定范围内的随机符点数，两个参数其中一个是上限，一个是下限。如果 a > b，则生成的随机数 n: b <= n <= a。如果 a <b， 则 a <= n <= b。

```
print random.uniform(10, 20)
print random.uniform(20, 10)
#---- 结果（不同机器上的结果不一样）
#18.7356606526
#12.5798298022
print random.uniform(10, 20)
print random.uniform(20, 10)
#---- 结果（不同机器上的结果不一样）
#18.7356606526
#12.5798298022
```

random.randint

random.randint()的函数原型为：random.randint(a, b)，用于生成一个指定范围内的整数。其中参数 a 是下限，参数 b 是上限，生成的随机数 n: a <= n <= b

```
print random.randint(12, 20)  #生成的随机数 n: 12 <= n <= 20
print random.randint(20, 20)  #结果永远是 20
#print random.randint(20, 10)  #该语句是错误的。下限必须小于上限。
print random.randint(12, 20)  #生成的随机数 n: 12 <= n <= 20
print random.randint(20, 20)  #结果永远是 20
```

```
#print random.randint(20, 10)  #该语句是错误的。下限必须小于上限。
```

random.randrange

random.randrange 的函数原型为：random.randrange([start], stop[, step])，从指定范围内，按指定基数递增的集合中 获取一个随机数。如：random.randrange(10, 100, 2)，结果相当于从[10, 12, 14, 16, ... 96, 98]序列中获取一个随机数。random.randrange(10, 100, 2)在结果上与 random.choice(range(10, 100, 2) 等效。

random.choice

random.choice 从序列中获取一个随机元素。其函数原型为：random.choice(sequence)。参数 sequence 表示一个有序类型。这里要说明 一下：sequence 在 python 不是一种特定的类型，而是泛指一系列的类型。list, tuple, 字符串都属于 sequence。有关 sequence 可以查看 python 手册数据模型这一章，也可以参考：http://www.17xie.com/read-37422.html 。下面是使用 choice 的一些例子：

```
print random.choice("学习 Python")
print random.choice(["JGood", "is", "a", "handsome", "boy"])
print random.choice(("Tuple", "List", "Dict"))
print random.choice("学习 Python")
print random.choice(["JGood", "is", "a", "handsome", "boy"])
print random.choice(("Tuple", "List", "Dict"))
```

random.shuffle

random.shuffle 的函数原型为：random.shuffle(x[, random])，用于将一个列表中的元素打乱。如:p = ["Python", "is", "powerful", "simple", "and so on..."]

```
random.shuffle(p)
print p
#---- 结果（不同机器上的结果可能不一样。）
#['powerful', 'simple', 'is', 'Python', 'and so on...']
p = ["Python", "is", "powerful", "simple", "and so on..."]
random.shuffle(p)
print p
#---- 结果（不同机器上的结果可能不一样。）
#['powerful', 'simple', 'is', 'Python', 'and so on...']
```

random.sample

random.sample 的函数原型为：random.sample(sequence, k)，从指定序列中随机获取指定长度的片断。sample 函数不会修改原有序列。

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
slice = random.sample(list, 5)  #从 list 中随机获取 5 个元素，作为一个片断返回
print slice
print list #原有序列并没有改变。
```

```
list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
slice = random.sample(list, 5)  #从 list 中随机获取 5 个元素，作为一个片断返回
print slice
print list #原有序列并没有改变。
```

上面这些方法是 random 模块中最常用的，在 Python 手册中，还介绍其他的方法。感兴趣的朋友可以通过查询 Python 手册了解更详细的信息。

python 的内嵌 time 模板翻译及说明

一、简介

time 模块提供各种操作时间的函数

说明：一般有两种表示时间的方式:

第一种是时间戳的方式(相对于 1970.1.1 00:00:00 以秒计算的偏移量),时间戳是惟一的

第二种以数组的形式表示即(struct_time),共有九个元素，分别表示，同一个时间戳的 struct_time 会因为时区不同而不同

```
year (four digits, e.g. 1998)
month (1-12)
day (1-31)
hours (0-23)
minutes (0-59)
seconds (0-59)
weekday (0-6, Monday is 0)
Julian day (day in the year, 1-366)
DST (Daylight Savings Time) flag (-1, 0 or 1) 是否是夏令时
If the DST flag is 0, the time is given in the regular time zone;
if it is 1, the time is given in the DST time zone;
if it is -1, mktime() should guess based on the date and time.
夏令时介绍：http://baike.baidu.com/view/100246.htm
UTC 介绍：http://wenda.tianya.cn/wenda/thread?tid=283921a9da7c5aef&clk=wttpcts
```

二、函数介绍

1.asctime()

asctime([tuple]) -> string

将一个 struct_time(默认为当时时间)，转换成字符串

```
Convert a time tuple to a string, e.g. 'Sat Jun 06 16:26:11 1998'.
When the time tuple is not present, current time as returned by localtime()
is used.
```

2.clock()

clock() -> floating point number
该函数有两个功能，
在第一次调用的时候，返回的是程序运行的实际时间；
以第二次之后的调用，返回的是自第一次调用后,到这次调用的时间间隔

示例：
view plaincopy to clipboardprint?

```
import time
if __name__ == '__main__':
    time.sleep(1)
    print "clock1:%s" % time.clock()
    time.sleep(1)
    print "clock2:%s" % time.clock()
    time.sleep(1)
    print "clock3:%s" % time.clock()
import time
if __name__ == '__main__':
    time.sleep(1)
    print "clock1:%s" % time.clock()
    time.sleep(1)
    print "clock2:%s" % time.clock()
    time.sleep(1)
    print "clock3:%s" % time.clock()
```

输出：

```
clock1:3.35238137808e-006
clock2:1.00004944763
clock3:2.00012040636
```

其中第一个 clock 输出的是程序运行时间
第二、三个 clock 输出的都是与第一个 clock 的时间间隔

3.sleep(...)
sleep(seconds)
线程推迟指定的时间运行，经过测试，单位为秒，但是在帮助文档中有以下这样一句话，这关是看不懂
“The argument may be a floating point number for subsecond precision.”

4.ctime(...)
ctime(seconds) -> string
将一个时间戳(默认为当前时间)转换成一个时间字符串
例如：
time.ctime()
输出为：'Sat Mar 28 22:24:24 2009'

5.gmtime(...)

gmtime([seconds]) -> (tm_year, tm_mon, tm_day, tm_hour, tm_min,tm_sec, tm_wday, tm_yday, tm_isdst)

将一个时间戳转换成一个UTC时区(0时区)的struct_time，如果seconds参数未输入，则以当前时间为转换标准

6.localtime(...)

localtime([seconds]) -> (tm_year,tm_mon,tm_day,tm_hour,tm_min,tm_sec,tm_wday,tm_yday,tm_isdst)

将一个时间戳转换成一个当前时区的struct_time，如果seconds参数未输入，则以当前时间为转换标准

7.mktime(...)

mktime(tuple) -> floating point number

将一个以struct_time转换为时间戳

8.strftime(...)

strftime(format[, tuple]) -> string

将指定的struct_time(默认为当前时间)，根据指定的格式化字符串输出

python中时间日期格式化符号：

%y 两位数的年份表示（00-99）

%Y 四位数的年份表示（000-9999）

%m 月份（01-12）

%d 月内中的一天（0-31）

%H 24小时制小时数（0-23）

%I 12小时制小时数（01-12）

%M 分钟数（00=59）

%S 秒（00-59）

%a 本地简化星期名称

%A 本地完整星期名称

%b 本地简化的月份名称

%B 本地完整的月份名称

%c 本地相应的日期表示和时间表示

%j 年内的一天（001-366）

%p 本地A.M.或P.M.的等价符

%U 一年中的星期数（00-53）星期天为星期的开始

%w 星期（0-6），星期天为星期的开始

%W 一年中的星期数（00-53）星期一为星期的开始

%x 本地相应的日期表示

%X 本地相应的时间表示

%Z 当前时区的名称

%% %号本身

9.strptime(...)

strptime(string, format) -> struct_time

将时间字符串根据指定的格式化符转换成数组形式的时间

例如：

2009-03-20 11:45:39 对应的格式化字符串为：%Y-%m-%d %H:%M:%S

Sat Mar 28 22:24:24 2009 对应的格式化字符串为：%a %b %d %H:%M:%S %Y

10.time(...)

time() -> floating point number

返回当前时间的时间戳

三、疑点

1.夏令时

在 struct_time 中，夏令时好像没有用，例如

```
a = (2009, 6, 28, 23, 8, 34, 5, 87, 1)
b = (2009, 6, 28, 23, 8, 34, 5, 87, 0)
```

a 和 b 分别表示的是夏令时和标准时间，它们之间转换为时间戳应该相关 3600，但是转换后输出都为 646585714.0

四、小应用

1.python 获取当前时间

time.time() 获取当前时间戳

time.localtime() 当前时间的 struct_time 形式

time.ctime() 当前时间的字符串形式

2.python 格式化字符串

格式化成 2009-03-20 11:45:39 形式

```
time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
```

格式化成 Sat Mar 28 22:24:24 2009 形式

```
time.strftime("%a %b %d %H:%M:%S %Y", time.localtime())
```

3.将格式字符串转换为时间戳

```
a = "Sat Mar 28 22:24:24 2009"
b = time.mktime(time.strptime(a,"%a %b %d %H:%M:%S %Y"))
```